

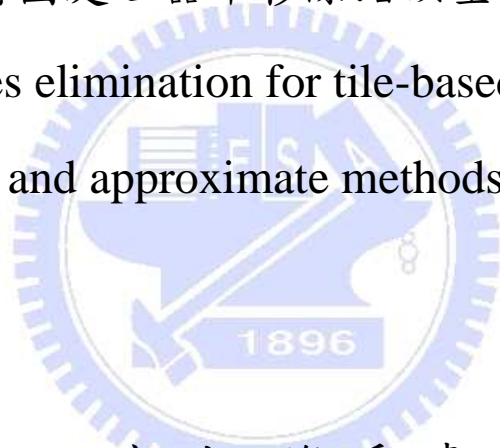
# 國立交通大學

多媒體工程研究所

碩士論文

在區塊著色繪圖處理器中移除錯誤重疊區塊之方法

False-overlap tiles elimination for tile-based rendering: exact  
and approximate methods



研究生：謝秀青

指導教授：單智君 林正中教授

中華民國九十八年六月

在區塊著色繪圖處理器中移除錯誤重疊區塊之方法

False-overlap tiles elimination: exact and approximate  
methods

研究生：謝秀青

Student : Hsiu-ching Hsieh

指導教授：單智君 林正中

Advisor : Jyh-Jiun Shann

Cheng-Chung Lin



Submitted to Institute of Multi-media  
College of Computer Science  
National Chiao Tung University  
in Partial Fulfillment of the Requirements  
for the Degree of  
Master  
in

Computer Science

June 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

# 在區塊著色繪圖處理器中移除錯誤重疊區塊之方法

學生：謝秀青

指導教授：單智君 林正中教授

國立交通大學多媒體工程研究所 碩士班

## 摘 要

在嵌入式裝置的圖形處理中，在進到 tile-binning 之前，overlap test 是一個很重要的步驟。Primitive 不管是任意大小、形狀、位置，都會有其 Bounding Box。然後 Bounding box 的面積相對於 primitive 而言，有 70% 是多餘的。在 tile-based rendering 中，辨識出在 bounding box 中沒有與 primitive 的重疊關係的 tile 且刪除它，可以節省 primitive list 的儲存空間大小及後續不必要的動作。現有的 false-overlap 偵測演算法不是過程繁瑣且不夠直覺，不然就是過於簡略，無法精確找出 false-overlap tile。在此，我們提出一些精確找出 false-overlap tile 的方法：Cross Product Test (CPT), Edge Walk (EW), Count X Ratio (CXR) 和 Approximation Method。設計重點在於，以較少的硬體設計出時間複雜度低的設計。為了提升效能，我們將 bounding box 切分成三個矩形，使得 primitive 的邊會是每個矩形的對角線且都有完整的數學公式來處理此區域內的 false-overlap tile。如此一來，也可以平行處理被切割的 primitive。

# False-overlap tiles elimination: exact and approximate methods

Student : Hsiu-ching Hsieh

Advisor : Jyh-Jiun Shann  
Cheng-Chung Li

Institute of Multi-media  
National Chiao-Tung University

## Abstract

In graphics processing, overlap test is a crucial step before tile-binning in tile-based rendering for embedded devices. An object in a frame is decomposed into primitives, triangles of different sizes, for processing. In tile-binning process, these triangular primitives are typically represented by bounding boxes. However, the bounding box of a primitive usually covers a significant number of tiles which are not overlapped by the primitive. These tiles are called false-overlap tiles and approximate 70% of the tiles of a bounding box. Therefore, in tile-based rendering, identifying and eliminating those false-overlap tiles in a bounding box to reduce both storage pressures in tile-binning and data accesses of external memory for rasterizer become inviting. Existing false-overlap detection algorithms are either too tedious to reduce computation or too rough to gain high coverage. In this paper, we propose three methods to eliminate all false-overlap tiles: Cross-Product Test (CPT), Edge-Walk Test (EWT), Counting X-Ratio (CXR) and approximation method. We partition the bounding box of a primitive into three rectangles at most according to the number of primitive vertices which are also the vertices of the bounding box. The edges of the primitive then become the diagonals of these rectangles, and false overlap detection becomes a well-formulated math processing. The false-overlap detection of these three rectangles may be processed in parallel to improve performance further. The proposed methods are tested using Doom3 and Quake4 for different screen sizes.

# Table of Contents

摘 要 .....	i
Abstract .....	ii
Table of Contents .....	v
List of Figures .....	vii
List of Tables .....	ix
<b>Chapter 1 Introduction .....</b>	<b>1</b>
<b>1.1 Research observation .....</b>	<b>1</b>
<b>1.2 Research motivation and objective .....</b>	<b>3</b>
<b>1.3 Organization of this thesis .....</b>	<b>4</b>
<b>Chapter 2 Background and Related work.....</b>	<b>5</b>
<b>2.1 Typical graphics pipeline .....</b>	<b>5</b>
<b>2.2 Tile-based rendering.....</b>	<b>6</b>
<b>2.2.1 Tile-based rendering pipeline .....</b>	<b>6</b>
<b>2.2.2 Data structures for Primitive lists .....</b>	<b>8</b>
<b>2.3 Primitive list Problem .....</b>	<b>8</b>
<b>2.4 Related works.....</b>	<b>9</b>
<b>2.4.1 Linear Edge Function Test (LET) .....</b>	<b>9</b>
<b>2.4.2 Iterative Division Test (IDT).....</b>	<b>11</b>
<b>Chapter 3 Design.....</b>	<b>13</b>
<b>3.1 Exact false-overlap detections .....</b>	<b>13</b>
<b>3.1.1 BBox Division.....</b>	<b>14</b>
<b>3.1.2 False-overlap Elimination by Exact False-Overlap Tile Detection             Algorithms.....</b>	<b>18</b>
<b>3.2 False-overlap Detection by Approximate Method.....</b>	<b>25</b>
<b>3.2.1 Vertexes Alignment.....</b>	<b>26</b>
<b>3.2.2 Boundary Expanding and Sub BBoxes.....</b>	<b>27</b>
<b>3.2.3 Elimination by Table Lookup.....</b>	<b>28</b>
<b>Chapter 4 Evaluation Results and Discussion.....</b>	<b>37</b>
<b>4.1 Evaluation environment.....</b>	<b>37</b>
<b>4.2 Test frame data .....</b>	<b>39</b>
<b>4.3 Simulation results .....</b>	<b>41</b>
<b>4.3.1 Correct Rates of Different Entry Size of Differential Table for our             Approximation Method.....</b>	<b>41</b>
<b>4.3.2 Storage Size of Primitive Lists.....</b>	<b>43</b>
<b>4.3.3 Time complexity.....</b>	<b>49</b>

<b>Chapter 5</b>	<b>Conclusion and Future Work</b>	51
5.1	Conclusion	51
5.2	Future work	52
<b>References</b>		53
<b>Appendix A</b>	<b>Simulation Test Frame Images</b>	55



# List of Figures

Figure1-1	The relation of the primitive, tiles and BBox in the BBox test .....	2
Figure1-2	The benchmarks observed are Doom 3 and Quake 4. ....	3
Figure1-3	Average percentage of false-overlap tiles list in a BBox test .....	3
Figure 2-1	Typical 3D graphics pipeline .....	5
Figure 2-2	Tile-based rendering pipeline .....	6
Figure 2-3	Tile binning process.....	7
Figure 2-4	Linked-list implementations of primitive lists .....	8
Figure 2-5	Relationship of Tile and Primitive list.....	9
Figure 2-6	Edge function for a point and a line .....	10
Figure 2-7	Triangle to tile test using linear function.....	10
Figure 2-8	Example for Iterative Division Test.....	11
Figure 2-9	Algorithm of Iterative Division Test.....	12
Figure 3-1	Processing flow of exact method.....	13
Figure 3-2	Partition of primitive <i>ABC</i> .....	14
Figure 3-3	Edge of primitive divides the rectangle into two triangular regions.....	15
Figure 3-4	Illustration and algorithm for distinguishing false-overlap region and preserved region	16
Figure 3-5	Algorithm for determining the number of filter units required .....	16
Figure 3-6	Flow chart for determining the number of filter units required.....	17
Figure 3-7	Cross Product Test (CPT) for detecting all possible false-overlap tiles. ....	19
Figure 3-8	CPT Algorithm .....	19
Figure 3-9	CPT hardware design.....	20
Figure 3-10	Optimal Bresenham's line algorithm.....	21
Figure 3-12	Edge Walk Test (EWT) algorithm .....	22
Figure 3-13	EWT hardware design .....	23
Figure 3-14	Example of Counting X Ratio .....	24
Figure 3-15	Algorithm of Counting X Ratio (CXR).....	24
Figure 3-16	EWT hardware design .....	25
Figure 3-17	Flowchart of approximate false-overlap detection .....	26
Figure 3-18	Cases analysis of vertex alignment.....	27
Figure 3-20	Using the width and height of a sub BBox to look up the differential table .....	28
Figure 3-21	Example of eliminating false-overlap tiles with differential table .....	29
Figure 3-22	Example of differential table reduction. ....	30
Figure 3-23	Difference of false-overlap tiles between adjacent rows.....	32
Figure 3-24	Circuit in front of Converter.....	33
Figure 3-25	Direction for eliminating false-overlap tiles.....	34

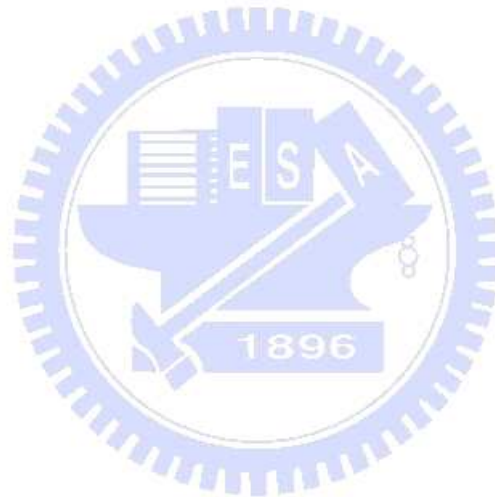
Figure 3-26	Example of eliminating rows larger the differential pattern.....	35
Figure 4-1	simulation flow and ATTILA architecture [13] .....	38
Figure 4-2	Frame 30 in DOOM3.....	38
Figure 4-3	Frame 30 in QUAKE4.....	39
Figure 4-4	Correct rate with entry size for DOOM3.....	42
Figure 4-5	Correct rate with entry size for QUAKE4.....	42
Figure 4-6	Amount of primitive lists for Doom3 .....	44
Figure 4-7	Amount of primitive lists for Quake4 .....	44
Figure 4-8	Number of Records in Primitive Lists with Iterative Division Test .....	45
Figure 4-9	Correct Rate of Iterative Division Test.....	45
Figure 4-10	Correct rate of primitive lists for DOOM3 .....	46
Figure 4-11	Correct rate of primitive lists for QUAKE4 .....	47
Figure 4-12	Reduction Rate of Primitive Lists for DOOM3 .....	48
Figure 4-13	Reduction Rate of Primitive Lists for QUAKE4.....	48
Figure 5-1	Use pixel pattern to eliminate false-overlap tiles .....	52

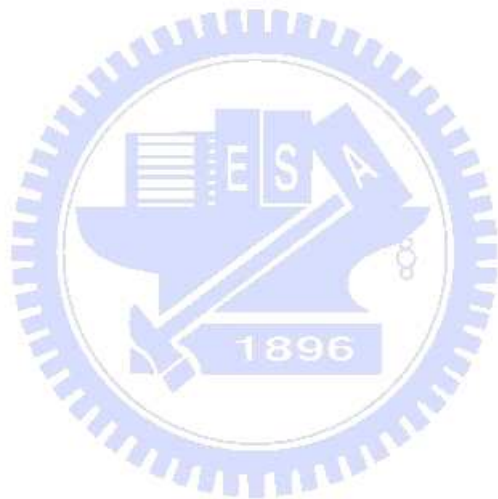




# List of Tables

Table 3-1	Error of different table entry.....	31
Table 3-2	Inputs and output of converter.....	34
Table 4-1	Statistics of test frames on Doon3 .....	40
Table 4-2	Statistics of test frames on Quake 4 .....	40
Table 4-3.	Average number of different operations per right triangle of each algorithm in various tests.....	40
Table 4-3	Time complexity of our methods and relate work.....	49





# Chapter 1 Introduction

3D graphic applications in embedded systems such as 3D games [1], personal navigation devices, graphical user interface, etc., become more and more popular in recent years. As we knew, the embedded systems are designed for some specific applications. Accordingly, the system designers usually want to reduce the costs by limiting system resources. However, the 3D graphic applications in embedded systems become more complex than before. Therefore, the trade off between performance, power, and storage of 3D graphic processing in such systems becomes an important issue.

A promising technique called tile-based rendering [2] has been widely use in those resource-limited graphic processing environments like ARM Mali [3], PowerVR SGX [4], and ATi Imageon 2380 [5]. Instead of rendering a full frame in one pass, this technique divides screen into many small blocks called tiles and rendering tile by tile. Typically, tile size is 32x32 pixels, such that we can use less than 10KBytes for frame buffer and Z-buffer to store runtime information in rendering a tile. Due to this low runtime storage requirement, we can employ a small on-chip memory to render a scene instead of a large off-chip frame buffer and Z-buffer. Localized runtime storage can greatly reduces the external memory traffic in GPU. However, this technique requires extra buffers called scene buffer to store all primitives' data and each tile has a corresponding primitive list to record which primitives should be rendered in this tile. Then the primitives will be sent to renderer in per tile basis when rendering in progress.

## 1.1 Research observation

To render a tile of the scene, the tile-based rendering needs the information of the primitives which overlap with the tile. In other words, these primitives have to be stored into

the correspondent primitive list. The most commonly used method for a primitive to determine the tiles overlapped with it is Bounding Box (BBox) test [4,6-8] as shown in Fig 1-1(a). However, there are false-overlap tiles which overlapped with BBox only but not the primitive in the BBox test as Fig 1-1(b) shown. The primitive list of a false-overlap tile will also keep the information of the primitive. After accessing the primitive list from the external memory to render the tile, rasterizer will find out that the primitive does not overlap with the tile, i.e., the information of the primitive in the primitive list is redundant. Therefore, if we can detect a false-overlap tile before inserting the primitive information into the corresponding primitive list, we could both reduce the storage size of the primitive list and the data accesses of external memory.

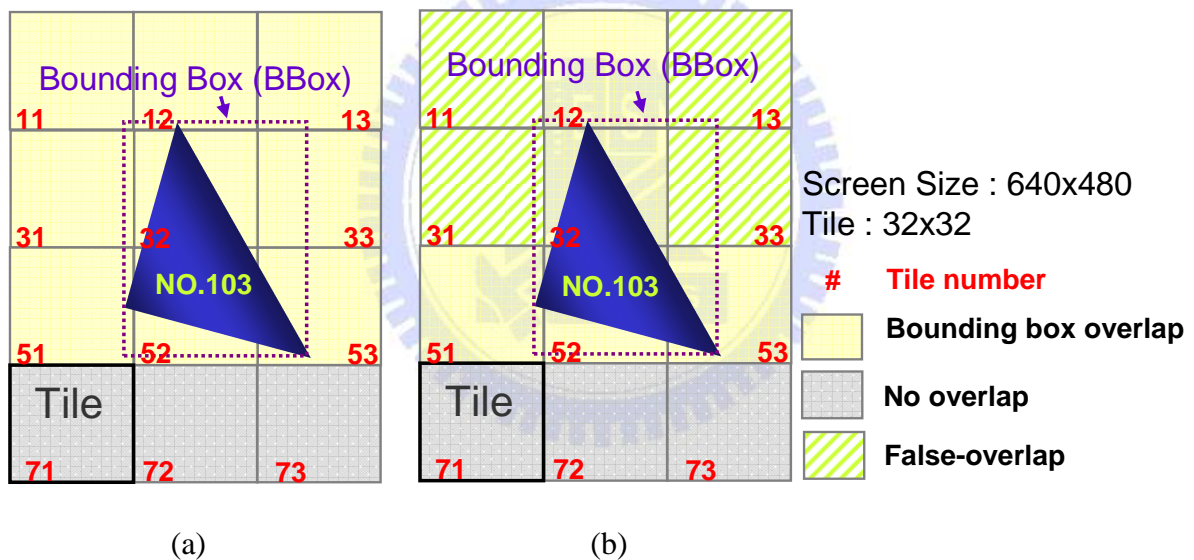


Figure1-1 The relation of the primitive, tiles and BBox in the BBox test.

The benchmarks observed are Doom3 and Quake4 as shown in Fig1-2 and the average percentages of frame 30, 60, 90, 120, 150 of false-overlap primitive list in BBOX test for various screen sizes is growing up with the resolution as shown in Fig 1-3. The reason is that when the resolution is getting higher, the primitive and the bounding box become larger, and thus the false-overlap has more change to happen. Simulation results show that there are 30 ~ 65% false-overlap primitive lists in Doom3 and 58 ~ 85% in Quake4 while applying traditional BBox test.

If we can find out the false-overlap tiles early and avoid inserting the primitive information to the primitive list, we can reduce the amount of the primitive list and data traffic of memory access for rasterizer.



Figure1-2 The benchmarks observed are Doom 3 and Quake 4.

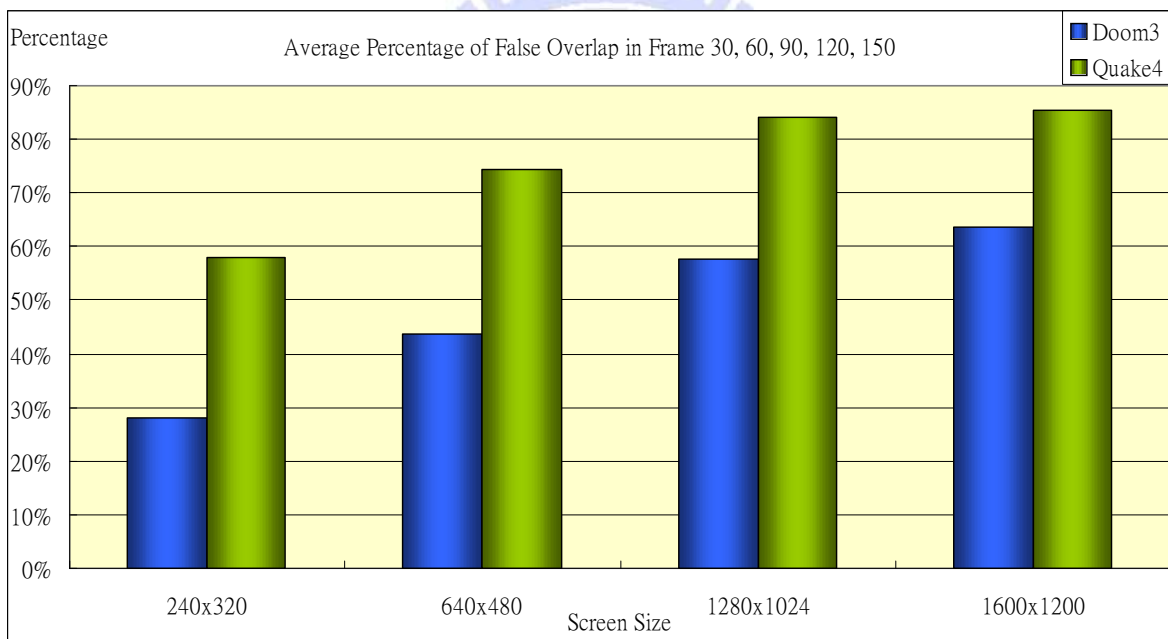


Figure1-3 Average percentage of false-overlap tiles list in a BBox test

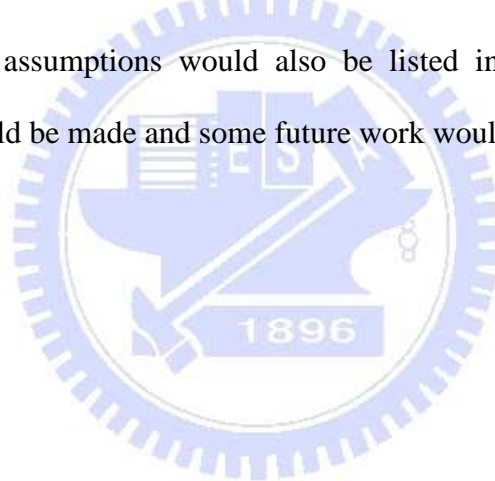
## 1.2 Research motivation and objective

As mentioned above, there is a significant amount of false-overlap tiles in bounding boxes. Hence, in tile-based rendering, identifying and eliminating those false-overlap tiles in a bounding box to reduce both storage pressure in tile-binning and data accesses of external

memory for rasterizer become inviting. Existing false-overlap detection algorithms are either too tedious to reduce computation or too rough to gain high coverage. Therefore, we propose four methods for false-overlap detection in tile-based rendering by the relationship of the primitive's edges and the tiles.

### **1.3 Organization of this thesis**

The main chapters of this thesis are organized as follows: In chapter 2, we would provide background knowledge for tile-based rendering, and related works would be introduced. In chapter 3, we would present four different approaches of our design and a plain evaluation of the four methods. Chapter 4 would demonstrate the simulation technique and results of this work; some environment assumptions would also be listed in this chapter. And finally, Chapter 5, a summary would be made and some future work would be proposed.



# Chapter 2 Background and Related work

In this chapter, we will give an overview of typical graphics pipeline. Then, we will introduce the tile-based rendering; explain the differences between these two different GPU implementations. Also, the inefficiency of memory usage in tile-based rendering will be discussed. At the end of this chapter, we will present the details of two previous work related to this problem.

## 2.1 Typical graphics pipeline

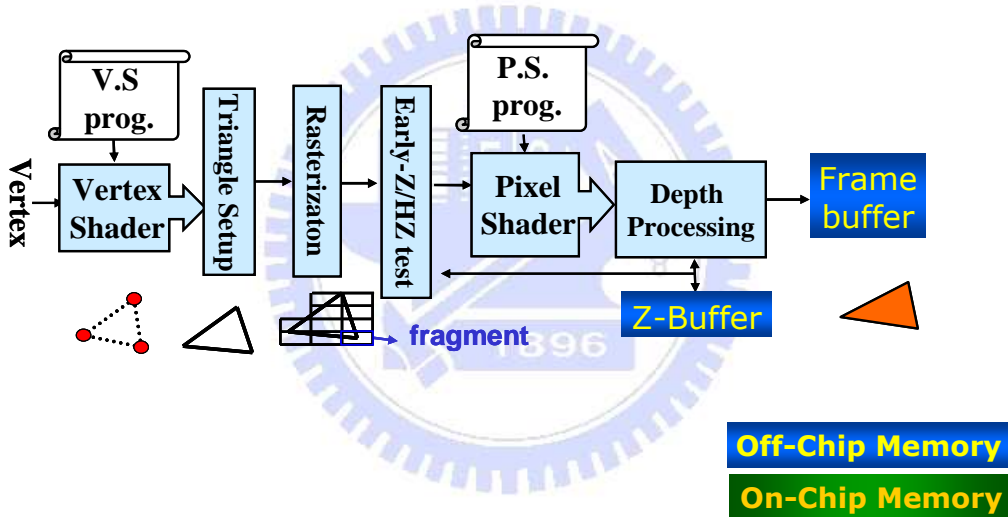


Figure 2-1 Typical 3D graphics pipeline

Typical 3D graphics pipeline can be showed as Figure 2-1. Each object in a 3D scene may be composed of many primitives, typically triangles. And each triangle consists of three vertices. The graphics pipeline will perform coordinate transformation on each vertex from object space to 3D scene space, and finally into screen space by vertex shader. And then, the triangle setup will assemble vertices into primitives. In rasterization stage, the primitive will be rasterized into many fragments according to its screen coordinates. These fragments will be tested by Early-Z or Hierarchical-Z test to filter out invisible fragments as soon as possible to reduce the workload in pixel shader and Z-test. These fragments that passed Early-Z or

Hierarchical-Z test will be sent to pixel shader to perform color shading and texture filtering. After fragment shading process in pixel shader, the final Z-test will perform on each shaded fragment to see if it should be displayed on the screen or not and then send to frame buffer and update corresponding value in Z-buffer according to the test result.

In this process, both Z-test and frame buffer are external memories which means that access these two buffers will cause extra latencies. As the 3D scenes become complex, there are more than ten times of visible fragments that need to access these two buffers since primitives are not process by any specific order and cause lot of external memory traffic.

## 2.2 Tile-based rendering

In this section, we will introduce the basis of tile-based rendering and its corresponding data structures. And finally discuss some observations and problems.

### 2.2.1 Tile-based rendering pipeline

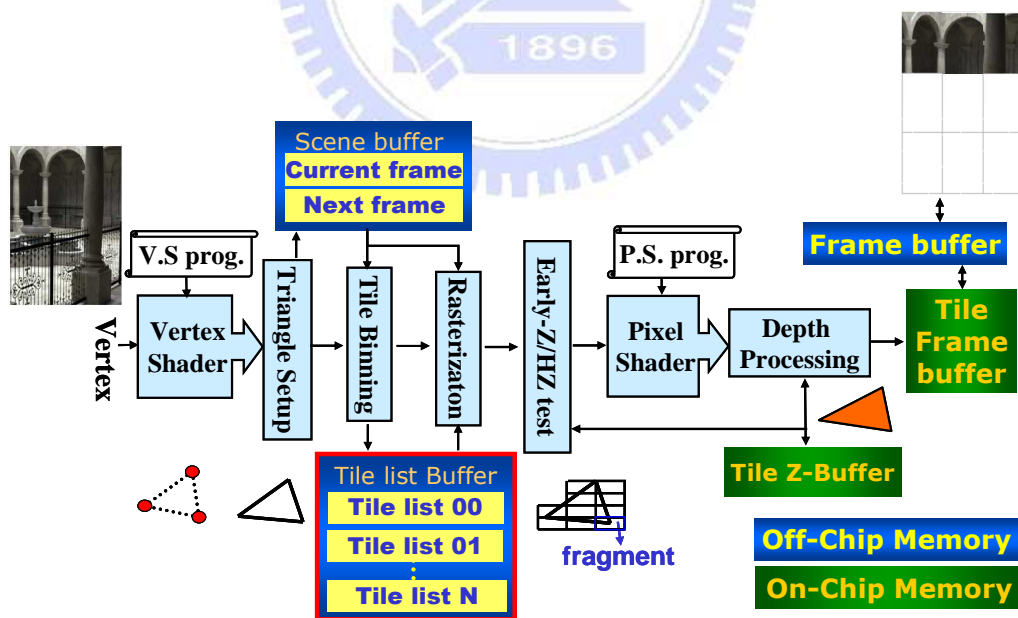


Figure 2-2 Tile-based rendering pipeline

As for the tile-based rendering GPU, instead of rendering a full frame at a time, this technique render a small region of frame, called a tile which is typically 32x32 pixels, one by



one. According to this characteristic, the temporary storage such as the Z-buffer and the frame buffer can be easily built in a chip, and thus significantly reduce the external memory traffic.

Figure 2-2 shows the diagram of the tile-based rendering pipeline. The process before triangle setup is exactly the same as that of the typical graphics pipeline. After triangle setup, the data of the transformed primitives will be stored in an extra storage called scene buffer. Also, each tile on screen has a corresponding external storage called primitive list which records the primitives rendered in this tile. After storing all primitives of a frame into the scene buffer, the tile binning process will be performed. As Figure 2-3 shows, the tile binning process will begin with the bounding box test which is formed by the primitive's maximum X and Y and minimum X and Y values of its transformed vertices' coordinates. This bounding box will be used to check which tiles are covered by this bounding box. If a tile is covered by the bounding box, then the scene buffer address of this primitive will be recorded into the primitive list of the tile. After all primitives in this frame are sorted into tiles, then the rendering process will begin in tile base. The disadvantage of this method is that the pixel process must start after all primitives in current frame have been sorted into tiles. Fortunately, this latency may be hidden by doubling scene buffer and primitive lists to process multiple frames simultaneously.

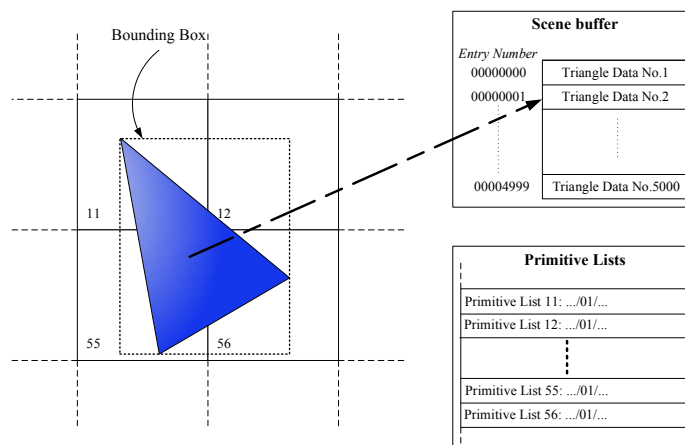


Figure 2-3 Tile binning process

## 2.2.2 Data structures for Primitive lists

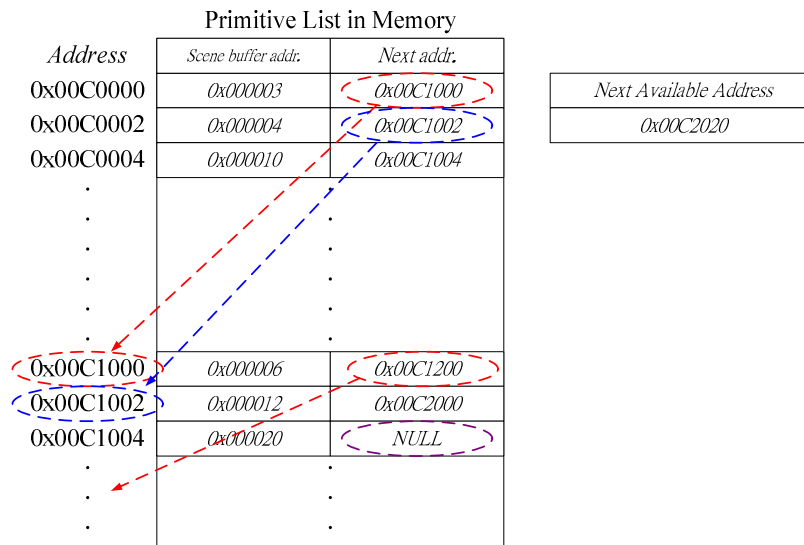


Figure 2-4 Linked-list implementations of primitive lists

Figure 2-4 shows the implementation of primitive lists. Each tile has a corresponding entry in the memory region. And each entry consists of two fields, one for the scene buffer address of a primitive and the other for the next record address. This implementation can ensure that no internal fragmentation in each list, but storage redundancy is very serious since it records every data with a corresponding next address. If a NULL is found in the next address filed, for example, 0x00C1004 in Figure 2-4, it means that the record is the end of the current primitive list.

Another way to implement the primitive list structure is using fixed-size storage in which every tile has a corresponding entry in memory with a fixed number of fields to record scene buffer addresses. Although this method is very efficiency in list retrieving, the internal fragmentation problem is very serious in it.

## 2.3 Primitive list Problem

Tile binning inserts the information of primitives into the primitive list of a tile to record which primitives overlap with the tile. Binning usually uses bounding box (BBox) test is

usually used in tile binning process to decide which tiles overlap with the primitive. In Fig 2-5, the purple dotted rectangle is the BBox of the primitive No.103. The information of primitive No.103 is inserted into the primitive lists of the tiles which overlap with the BBox. However, there are false-overlap tiles in BBox test. False-overlap tiles are those which overlap with the BBox only but not the primitive, as the green tiles in Figure 2-5. According to the traditional BBox test, tile binning also inserts the information of the primitive into the primitive lists of the false-overlap tiles. In this way, it also increases the data traffic for accessing external memory of rasterizer.

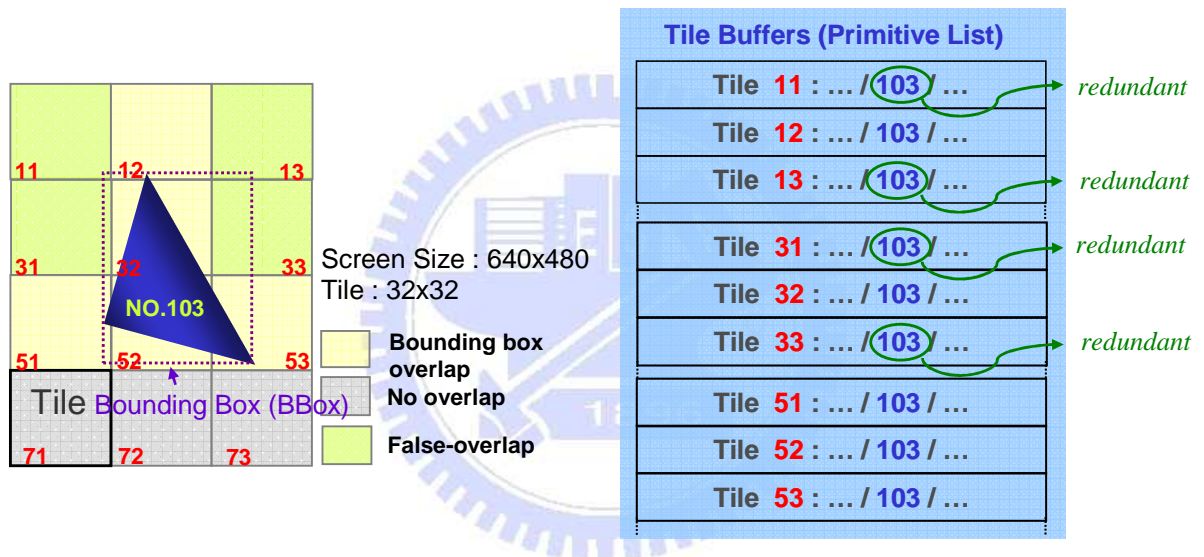


Figure 2-5 Relationship of Tile and Primitive list

## 2.4 Related works

### 2.4.1 Linear Edge Function Test (LET)

Antochi et al. proposed Linear Edge Test (LET) [9] to detect false-overlap tiles in 2004. LET employs edge function which is used to detect the relationship of a point and a line to filter out the false-overlap tiles [10].

Consider a 2D vector defined by two points A ( $X, Y$ ) and B ( $X+dX, Y+dY$ ) and a line  $L_{AB}$  that passes through the two points as shown in Fig 2-6. The edge function for a certain

point  $(x, y)$  is defined in the following:

$$E_{LAB}(x, y) = (x - X) \cdot dY - (y - Y) \cdot dX. \quad (2-1)$$

The edge function can be also written using an incremental form as follows:

$$E_{LAB}(x + \delta x, y + \delta y) = E_{LAB}(x, y) + \delta x \cdot dY - \delta y \cdot dX. \quad (2-2)$$

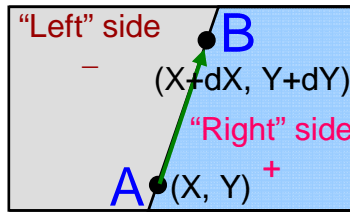


Figure 2-6 Edge function for a point and a line

The edge function can be used to determine the position of a point  $(x, y)$  relative to the line  $L_{AB}$  as follows:

- If  $E_{LAB}(x, y) > 0$  then the point is to the right of  $L_{AB}$
  - If  $E_{LAB}(x, y) = 0$  then the point is on  $L_{AB}$
  - If  $E_{LAB}(x, y) < 0$  then the point is to the left of  $L_{AB}$
- (2-3)

LET can be used to determine if a counter-clockwise oriented triangle  $T$ , defined by three vertices  $A(x_a, y_a)$ ,  $B(x_b, y_b)$ ,  $C(x_c, y_c)$ , intersects a square  $S$  defined by a center point  $CS(x_{cs}, y_{cs})$  and having width of 1. The determination equation are decided as follow:

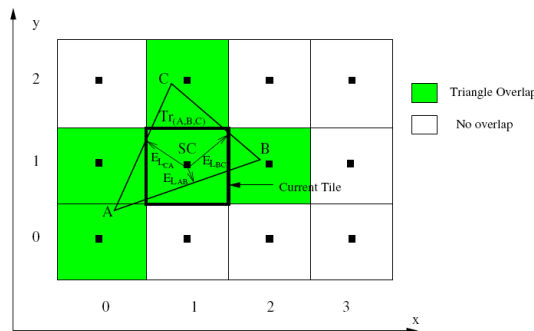


Figure 2-7 Triangle to tile test using linear function

$$\begin{aligned}
E_{LAB}(x_{CS}, y_{CS}) &\leq \frac{l}{2} \cdot (|x_B - x_A| + |y_B - y_A|) \\
E_{LBC}(x_{CS}, y_{CS}) &\leq \frac{l}{2} \cdot (|x_C - x_B| + |y_C - y_B|) \\
E_{LCA}(x_{CS}, y_{CS}) &\leq \frac{l}{2} \cdot (|x_A - x_C| + |y_A - y_C|)
\end{aligned}
\tag{2-4}$$

However, LET cannot eliminate all false-overlap tiles in the BBox and involves with a lot of floating multiplications and subtractions.

### 2.4.2 Iterative Division Test (IDT)

The method proposed by Intel [11] iteratively divides a primitive into several smaller triangles and makes the BBoxes of those triangles closer to the primitive. If the width of the BBox of the primitive is larger than a threshold, the primitive will be divided into smaller triangles by the middle points on the three vertices as shown in Fig 2-8.

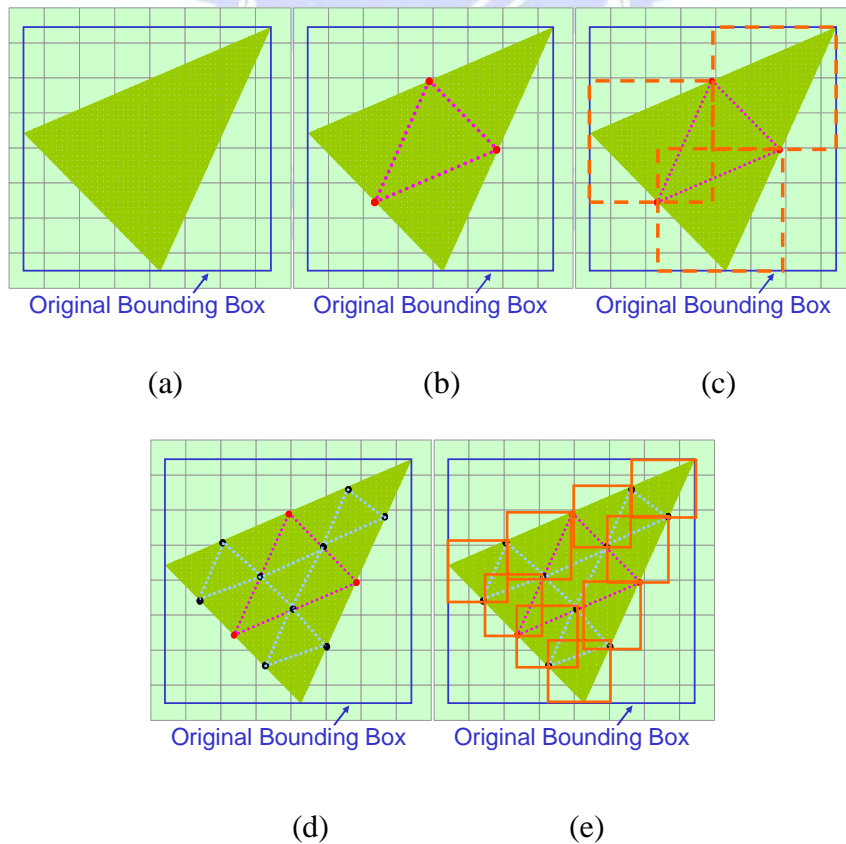


Figure 2-8 Example for Iterative Division Test

In Fig 2-8 (a), assume that the width of the BBox of the primitive is larger than the threshold, and then the primitive is divided into smaller triangles by the middle points of the three vertices as Fig 2-8 (b) shown. If the width of the BBox of these triangles is still larger than the threshold as shown in Fig 2-8 (c), these triangles are further divided into smaller triangle as Fig 2-8 (d) shown until the width of the BBox of the smaller triangle is smaller than threshold. The algorithm is described in Fig 2-9 [11].

```

{
  Compute new Reference Coordinate Point (RCP (current))
  if absolute value of (RCP (current) -RCP (previous))> threshold
  {
    Create a new Subdivision Record
    /* Do subdivision */
    Construct the axis aligned bounding box for the polygon
    Mark all tiles contained within the bounding box
    If either of the {x, y} dimensions of the bounding box are greater than a specified threshold,
    subdivide the polygon into four new polygon
      Compute the midpoints along each of the three line segments of the polygon
      Connect the three midpoints to form three new exterior polygons and one new interior polygon
      Generate new axis aligned bounding boxes for the exterior polygons
      Unmark any tiles from the original bounding box that are not contained in any of three new
      bounding boxes
      For each bounding box go to the dimension checking step above
    Store the bin assignment in the subdivision record
  }
  else
  {
    Resubmit polygon to the bins stored in subdivision record
  }
}

```

Figure 2-9 Algorithm of Iterative Division Test

Although the operation in the iterative division test is addition and the hardware cost is not high, it is necessary to process iteratively to get more precise result when the primitive is large. And the number of the sub BBoxes which increase processing time groups up with power of three. If the number of iterate is not suitable, there will be a lot of false-overlap tiles.

# Chapter 3 Design

In this chapter, our false-overlap detection methods are proposed. The objective of exact false-overlap detections is to eliminate all false-overlap tiles for different hardware resources and requirement. We also propose approximate method to remove false-overlap tiles roughly and quickly. This chapter is organized as follows: in section 3.1, exact false-overlap detections are introduced; in section 3.2 our approximate false-overlap detection is proposed.

## 3.1 Exact false-overlap detections

The processing flow of exact false-overlap detection is shown in Fig 3-1. After building BBox for a primitive, the BBox will be divided into rectangles. And then false-overlap elimination removes all false-overlap tiles in each rectangle. Finally, primitive listing inserts the information of the primitive into the primitive lists of the remaining tiles.

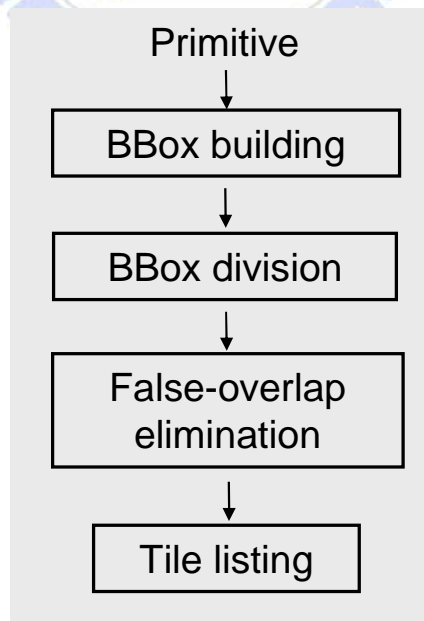


Figure 3-1 Processing flow of exact method

### 3.1.1 BBox Division

To eliminate false-overlap tiles precisely, we partition the BBox of a primitive into rectangles whose diagonal is one of the edge of the primitive. Moreover, the diagonal divides the rectangle into two right triangle regions. As shown in Fig.3-2, primitive  $ABC$  has BBox  $A EFG$  which is partitioned into rectangles  $ADCG$ ,  $IBFC$ , and  $AEBH$ . The edge  $AC$  of primitive  $ABC$  is the diagonal of the rectangle  $ADCG$  and divides the rectangle  $ADCG$  into two right triangle regions  $ACD$  and  $ACG$ , where  $ACG$  is the false-overlap region. Our algorithms are proposed to eliminate the false-overlap tiles in the false-overlap regions  $ACG$ ,  $ABE$ , and  $BCF$  in parallel. The other regions  $ACD$ ,  $ABH$ ,  $BCI$  are fully overlapped by the primitive. In Fig.3-2, the shadow tiles are the false-overlap tiles to be eliminated.

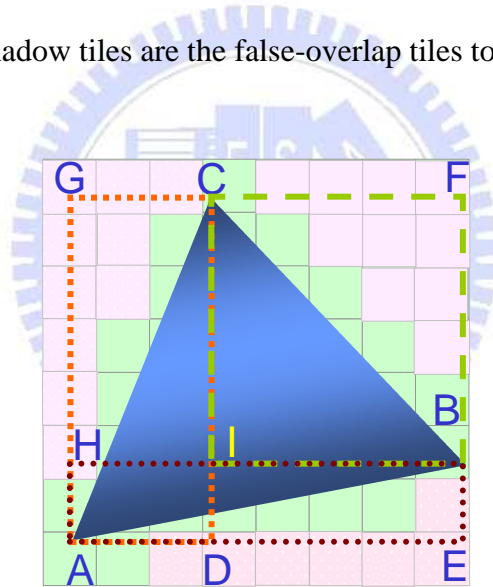


Figure 3-2 Partition of primitive  $ABC$

After BBox division, each primitive edge becomes the diagonal of a rectangle, and the diagonal divides the rectangle into two triangular regions, false-overlap region and preserved region, distinguishing by the third vertex of the primitive as shown in Fig. 3-3. With false-overlap and preserved regions, our exact methods can eliminate the false-overlap tiles in false-overlap region.

The diagonal contains two vertices of the primitive, the one which y coordinate is



smaller is called  $A$  and the other is called point  $C$ . The third vertex of the primitive is named point  $B$ . The one of other two vertex of the divided BBox which  $y$  coordinate is the same as point  $A$  is called point  $D$ , and the other one is called point  $E$  as the Fig. 3-4(a) shown. The algorithm for distinguishing false-overlap region and preserved region is shown in Fig. 3-4(a). If the values of  $x$  and  $y$  of directed segment  $\overrightarrow{AC}$  are larger than zero, and the cross product of  $\overrightarrow{AC}$  and  $\overrightarrow{AB}$  is larger than zero, the triangle region  $ACE$  in the Fig. 3-4(a) is false-overlap region. If values of  $x$  and  $y$  of directed segment  $\overrightarrow{AC}$  are larger than zero, and cross of  $\overrightarrow{AB}$  and  $\overrightarrow{AC}$  is smaller than zero, the triangle region  $ADC$  in the Fig. 3-4(a) is false-overlap region. If the value of  $x$  of directed segment  $\overrightarrow{AC}$  is smaller than zero,  $y$  of directed segment  $\overrightarrow{AC}$  large than zero and cross of  $\overrightarrow{AB}$  and  $\overrightarrow{AC}$  larger than zero, the triangle region  $ADE$  in the Fig. 3-4(a) is false-overlap region. If the value of  $x$  of directed segment  $\overrightarrow{AC}$  is smaller than zero,  $y$  of directed segment  $\overrightarrow{AC}$  large than zero and cross of  $\overrightarrow{AB}$  and  $\overrightarrow{AC}$  smaller than zero, the triangle region  $DEC$  in the Fig. 3-4(a) is false-overlap region.

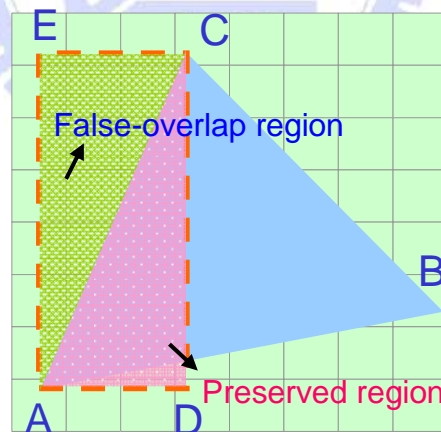


Figure 3-3 Edge of primitive divides the rectangle into two triangular regions

We design a filter unit which is the hardware to implement our false-overlap detection algorithm to distinguish the false-overlap tiles and eliminate them. We use the relation of tile and diagonal of the rectangle to develop three algorithms to remove the false-overlap tiles:

Cross Product Test (CPT), Edge Walk Test (EWT) and Counting X Ratio (CXR). The number of rectangles partitioned from a BBox of a primitive depends on the number of concurrents between the vertices of the primitive and that of the bounding box. In other word, the number of concurrent of primitive vertices and BBox vertices decide the number of filter units to eliminate the false-overlap tiles in parallel. The algorithm for determining the number of filter units required is shown in Fig.3-5.

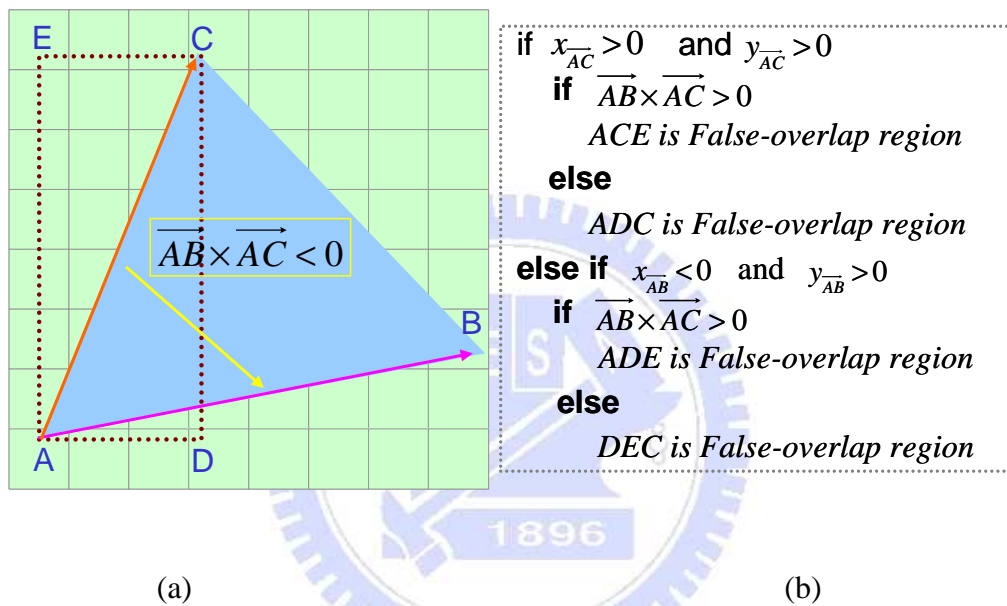


Figure 3-4 Illustration and algorithm for distinguishing false-overlap region and preserved region

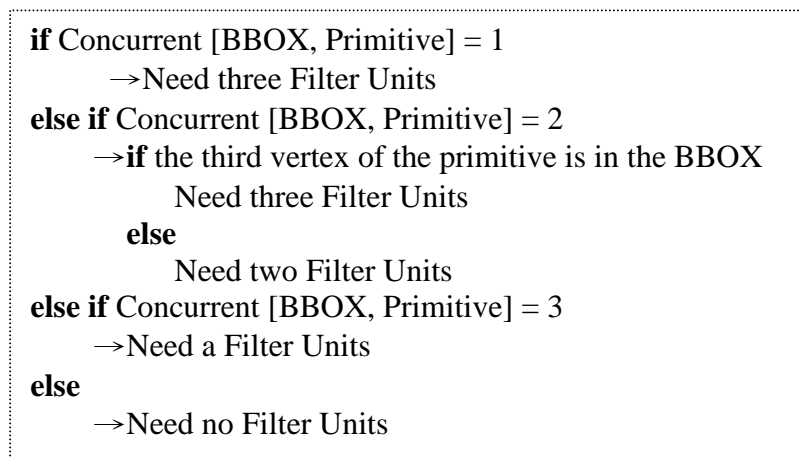


Figure 3-5 Algorithm for determining the number of filter units required

Fig. 3-6 is the flow chart for determining the number of filter units required. When the triangle setup outputs the triangle information to the tile binning unit, the tile binning unit makes bounding box for the primitive and counts many vertex of the primitive and the BBox are concurrent. According to the area of bounding box, tile binning can decide whether the false-overlap tiles should be eliminated or not. If the BBox has to be filtered out the false-overlap tiles in the false-overlap region, the concurrence will decide the number of rectangles partitioned from a BBox of a primitive to be processed. After eliminating all false-overlap tiles from a BBox, tile binning inserts the information of the primitive into the primitive lists of tile. The Min, Mid and Max are vertices of primitive ordered by y coordinate from small to large.

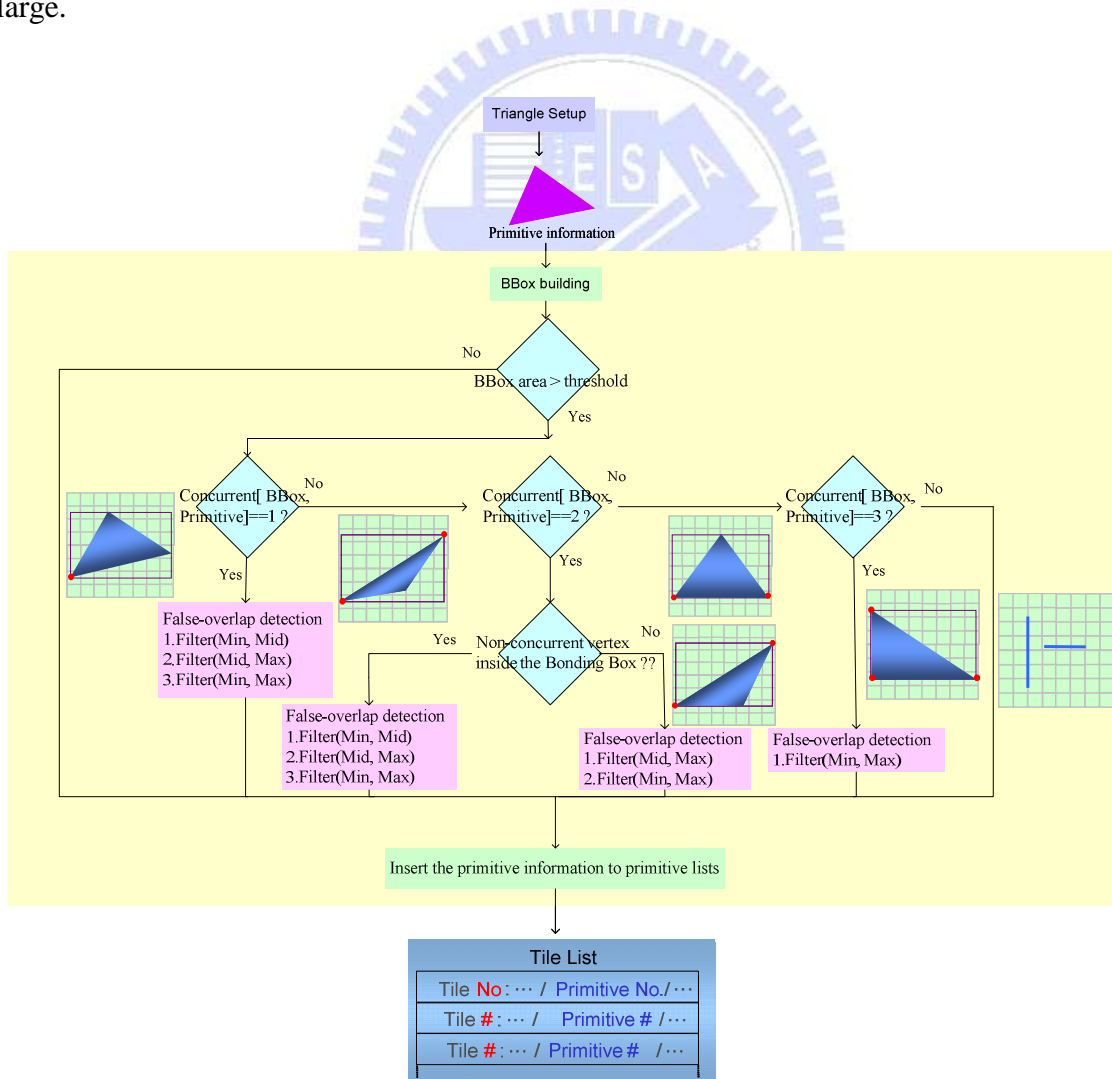


Figure 3-6 Flow chart for determining the number of filter units required

### 3.1.2 False-overlap Elimination by Exact False-Overlap Tile Detection Algorithms

In Fig. 3-7, 3-9, 3-12, we name one of the vertices on the diagonal in rectangle whose Y coordinate is the same as the right angle of the preserved region is called point *Start* (vertex *A* in these figures) and another vertex on the diagonal as point *End* (vertex *C* in these figures). The right angle of preserved region is called point  $R_p$  (vertex *D* in these figures), and right angle of false-overlap region is called point  $R_f$  (vertex *E* in these figures).

#### Algorithm for Cross Product Test (CPT)

CPT exams that a vertex of a tile which relative position to the tile is the same as point  $R_p$  to preserved region to decide the tile is false-overlap or not. If the tile is not false-overlap tile in the row, then CPT exam next tile.

For example, in Fig. 3-7, the relative position of point  $R_p$  to the preserved region ADC is lower right. We take the lower-right vertex of each tile in the false-overlap region for cross product test. Using the cross product to determine how line segments  $\overline{SE}$  (point *Start* to point *End*) and  $\overline{ST}$  (point *Start* to point *Tile vertex*) turn at point *Start*. We check whether the directed segment  $\overline{SE}$  is clockwise or counterclockwise relative to the directed segment  $\overline{SR}$  (from point *Start* to point *Right angle*) and  $\overline{ST}$ . If  $\overline{SE} \times \overline{SR}$  and  $\overline{SE} \times \overline{ST}$  have the same sign number, the primitive list of the tile can be inserted the information of the primitive. Otherwise,  $\overline{SE} \times \overline{SR}$  has different sign number to  $\overline{SE} \times \overline{ST}$ , the tile is a false-overlap tile. The algorithm is shown in Fig.3-8. And main advantage of algorithm CPT is that it needs integer multiplication and integer subtraction. Each cross product needs two multiplications and one subtraction. And the hardware design is show in Fig.3-9.

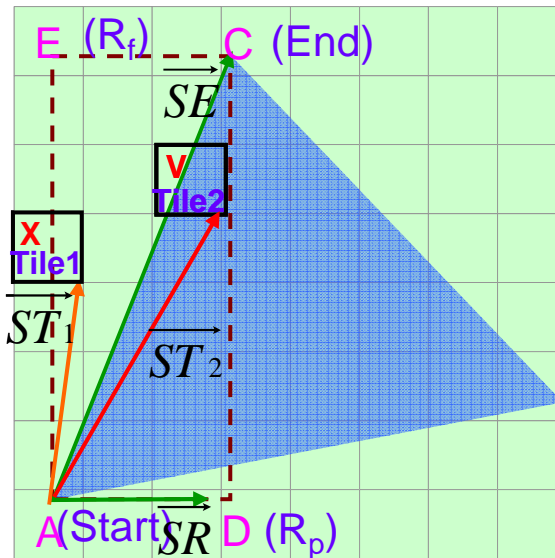


Figure 3-7 Cross Product Test (CPT) for detecting all possible false-overlap tiles.  $\vec{SE} \times \vec{SR} < 0$  and  $\vec{SE} \times \vec{ST}_1 > 0$  indicate that Tile1 false-overlaps with the primitive; whereas  $\vec{SE} \times \vec{SR} < 0$  and  $\vec{SE} \times \vec{ST}_2 < 0$  indicate that Tile2 overlaps with the primitive.

```

for row_next_Start to row_End
  for column_Start to column_End
    if ( $\vec{SE} \times \vec{SR} > 0$  and  $\vec{SE} \times \vec{ST} < 0$ ) or ( $\vec{SE} \times \vec{SR} < 0$  and  $\vec{SE} \times \vec{ST} > 0$ )
      the tile is false overlap tile for primitive
    else if ( $\vec{SE} \times \vec{SR} < 0$  and  $\vec{SE} \times \vec{ST} < 0$ ) or ( $\vec{SE} \times \vec{SR} > 0$  and  $\vec{SE} \times \vec{ST} > 0$ )
      the tile overlap with primitive
    else
      change to next row
  
```

Figure 3-8 CPT Algorithm

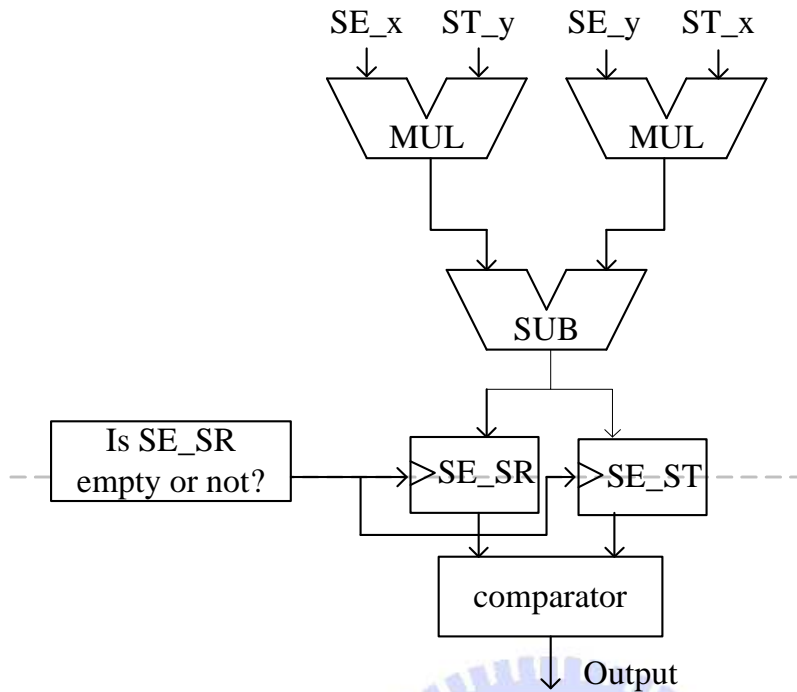


Figure 3-9 CPT hardware design

### Algorithm for Edge Walk Test (EWT)

In the algorithm Edge Walk Test, for each row, we eliminate false-overlap tiles from the column of point *Start* to the hypotenuse of preserved region. Here, we employ the optimal Bresenham's line algorithm [7] shown in Fig.3-10, to get x coordinate corresponding to the specific y coordinate on the hypotenuse of preserved region. The optimal Bresenham's line algorithm only uses integer addition, subtraction, and shift. It reduces hardware cost to get a value on linear edge without slope.

```

function optimal_Bresenham's_line (x0, x1, y0, y1)
  boolean steep := abs(y1 - y0) > abs(x1 - x0)
  if steep then
    swap(x0, y0)
    swap(x1, y1)
  if x0 > x1 then
    swap(x0, x1)
    swap(y0, y1)
  int deltax := x1 - x0
  int deltay := abs(y1 - y0)
  int error := deltax / 2
  int ystep
  int y := y0
  if y0 < y1 then ystep := 1 else ystep := - 1
  for x from x0 to x1
    if steep then plot(y,x) else plot(x,y)
    error := error - deltay
    if error < 0 then
      y := y + ystep
      error := error + deltax

```

Figure 3-10 Optimal Bresenham's line algorithm

If the y coordinate of point *Start* is smaller than that of point *End*, we let the specific y be the ceiling of y coordinate of point *Start*. Otherwise, specific y is the floor of y coordinate of point *Start*. We get the x coordinate corresponding to the specific y on the hypotenuse of preserved region, as  $\overline{AC}$  in Fig. 3-11, and then we can know the tile on the diagonal of rectangle in that row. The false-overlap tiles in each row are from the tile where column is the same as point *Start* to the tile on the diagonal of rectangle in that row. The algorithm is shown in Fig. 3-12. And the hardware design is show in Fig.3-13.

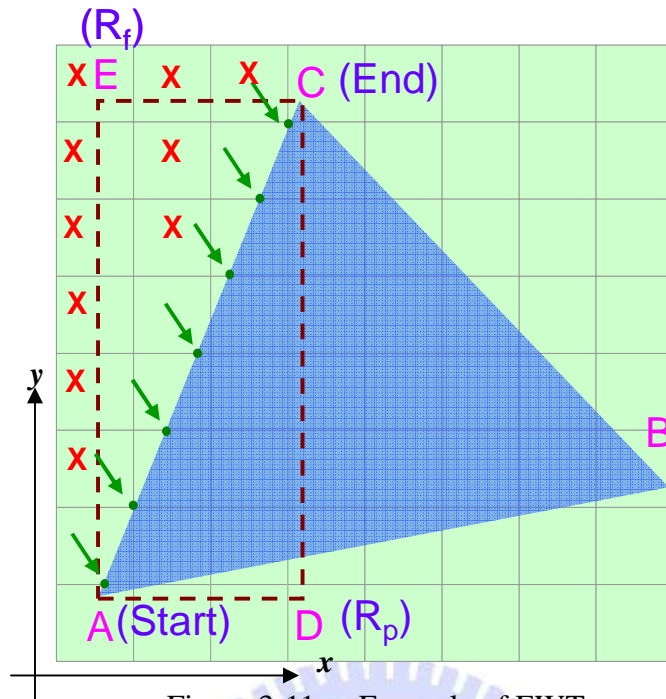


Figure 3-11 Example of EWT

```

if Start_y < End_y
    y = ceil (Start_y);
else
    y = floor (Start_y);
for row_next_Start to row_End
    x = Bresenham's line algorithm(y);
    compute the edge_tile of (x, y)
    for the column_same_Start to edge_tile
        delete the false overlap tile
    y += tile_height
  
```

Figure 3-12 Edge Walk Test (EWT) algorithm



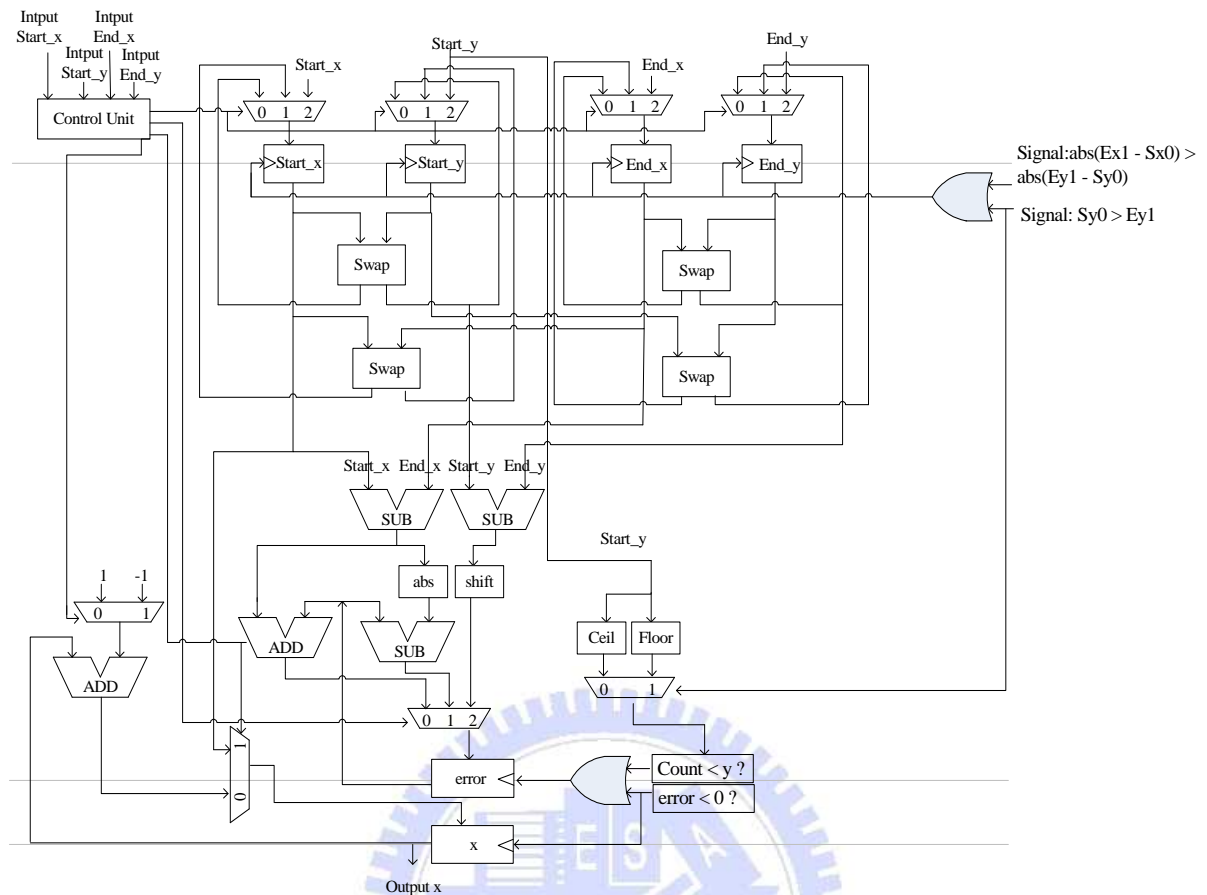


Figure 3-13 EWT hardware design

### Algorithm for Counting X Ratio (CXR)

In this algorithm, we accumulate the reciprocal of slope of hypotenuse of preserved region, as  $\overline{AC}$  in Fig. 3-14 to x ratio, for each row and the floor of ratio is the number of false-overlap tiles in that row. Here, initial x ratio is the ratio of the length on tile edge which is most close to point Start from point Start toward point End and is not covered by preserved region, as  $\overline{FG}$  in Fig 3-14, to tile width. Because there is no false-overlap tile in the row of point Start, we ignore computing the false-overlap tile in that row and start from the row next to row of point Start.

In Fig 3-14, the point F is the tile vertex which is most close to point Start from point Start toward point End and is not covered by preserved region, and the point G is an intersection point of the tile vertex which is most close to point Start from point Start toward point End and hypotenuse of preserved region. The initial x ratio is the ratio of  $\overline{FG}$  to tile width. The floor of x ratio is the number of false-overlap tiles. The algorithm is shown in Fig. 3-15. And the hardware design is show in Fig.3-16.

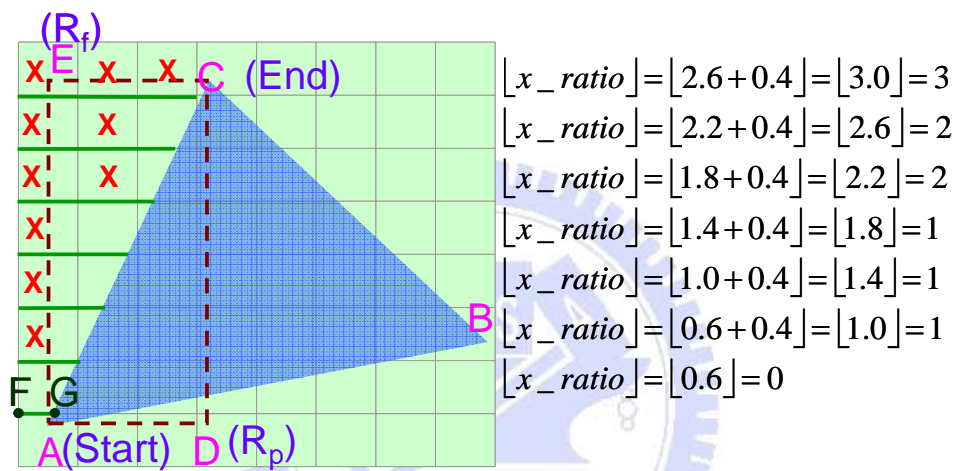


Figure 3-14 Example of Counting X Ratio

```

if Start_y < End_t
    y = ceil (Start_y);
else
    y = floor (Start_y);
x = Bresenham's line algorithm(y);
if Start_x < End_x
    x_ratio = (x % tile_width) / tile_width
else
    x_ratio = [ tile_width- (x % tile_width)] / tile_width
for row_next_Start to row_End
    delete_tile_num = ciel (x_ratio)
    for the column_same_Start to (column_same_Star+ delete_tile_num)
        delete the false overlap tile
    y += tile_height

x_ratio = reciprocal of slope_Start_End

```

Figure 3-15 Algorithm of Counting X Ratio (CXR)

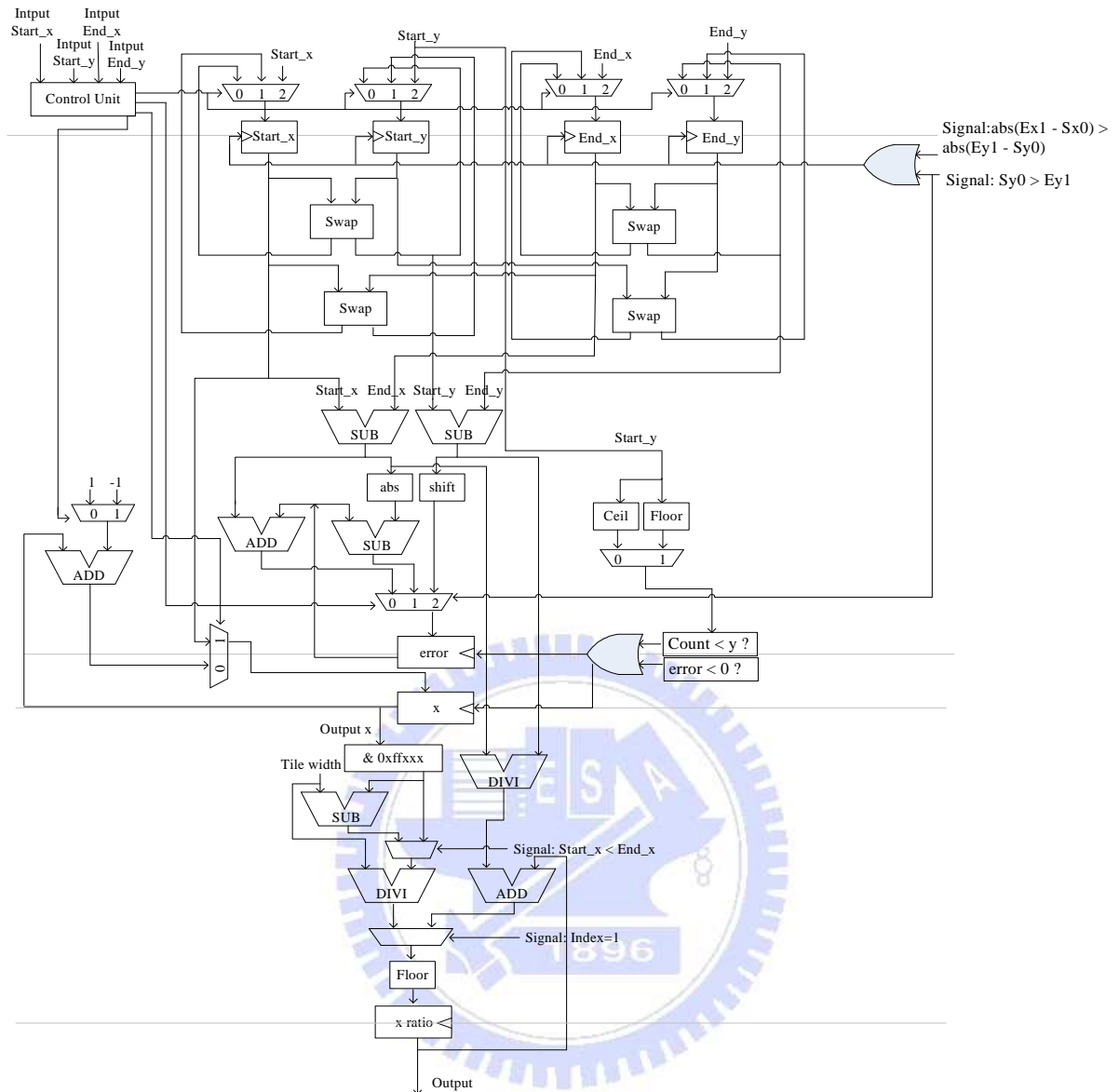


Figure 3-16 EWT hardware design

### 3.2 False-overlap Detection by Approximate Method

To avoid a lot of computations for eliminating false-overlap tiles, we try to use table lookup to reduce the computations. We observe that there are patterns for the numbers of false-overlap tiles in rows. If we could store the patterns of false-overlap tiles for difference in a table, we can eliminate the false-overlap tiles without complex computing.

Fig. 3-17 is the flow chart of approximate false-overlap detection. The difference of the flow chart to that of the exact methods is the vertex alignment and boundary expanding before BBox division. Vertex alignment aligns the vertices of a primitive to the tile vertices to the representative tile vertices, and then boundary expanding connects the representative tile vertices to form a new geometric figure whose area is larger than or equal to the primitive. BBox division divides the BBox of the primitive according to the expanding boundary and makes the expanded boundary be the diagonal of the sub BBox. After these approximations, we can eliminate false-overlap tiles in the false-overlap region by using the height and width of the sub BBox by table lookup.

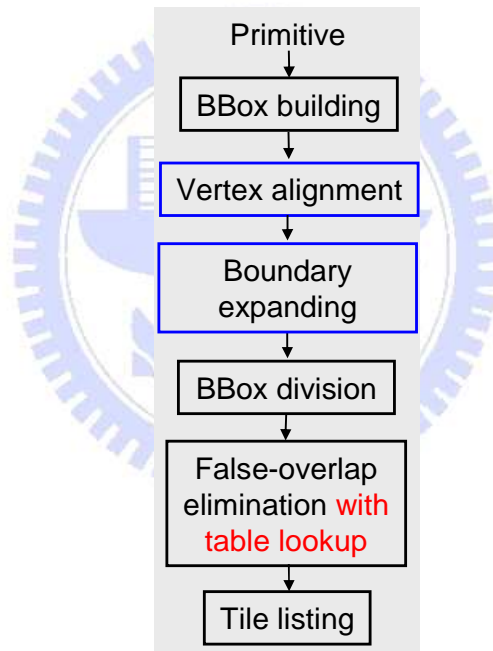


Figure 3-17 Flowchart of approximate false-overlap detection

### 3.2.1 Vertices Alignment

For reducing the computations, we align the vertices of a primitive to the tile vertex to representative tile vertices. According to the relation of vertices of a primitive and the number of tile vertices in the region of the extending edges (along the edge of primitive extend to outside of the primitive) of the primitive, we classify some cases of vertex alignment in

Fig.3-18.


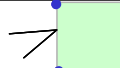


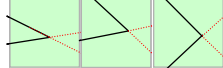
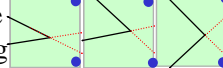
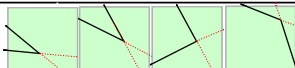
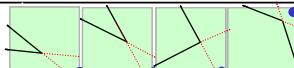
Case	Case description	How to choose the tile vertices to represent the vertex of the primitive ?
1	The vertex of the primitive is on the edge of a tile 	Choose both vertex of the edge of the tile 
2	The vertex of the primitive is on the vertex of a tile 	Choose the tile vertex overlapped with the vertex of the primitive 
3	There is no tile vertex in the region of the extending edges of the primitive 	Choose both vertex of the tile edge crossed by the extending edges of the primitive 
4	There are vertex in the region of the extending edges of the primitive 	Choose any one of the vertex in the extending edges of the primitive 

Figure 3-18 Cases analysis of vertex alignment

Case 1 is that the vertex of primitive is on the vertex of a tile, and the representative tile vertices are the vertices of primitive falls into. Class 2 is that when the vertices of primitive fall on the edge of tile , the representative tile vertices are the vertices on the edge of the tile. Case 3 is that there is no tile vertex in the region of extending edge at the vertex of the primitive. The representative tile vertices in case 3 are the tile vertices on the edge of tile which is crossed by the extending edge. In the case 4, there are vertices of tile in the reigon of extended edge at the vertex of primitive, and any one of these tile vertices can be representative tile vertices.

### 3.2.2 Boundary Expanding and Sub BBoxes

For each two vertices of the primitive, connect the representative tile vertices which are out of the edge at the same time.

Build a sub BBox for each expanded boundary of one of the edges of primitive, and let the expanded boundaries be the diagonals of the sub BBoxes as shown in the Fig.3-19.

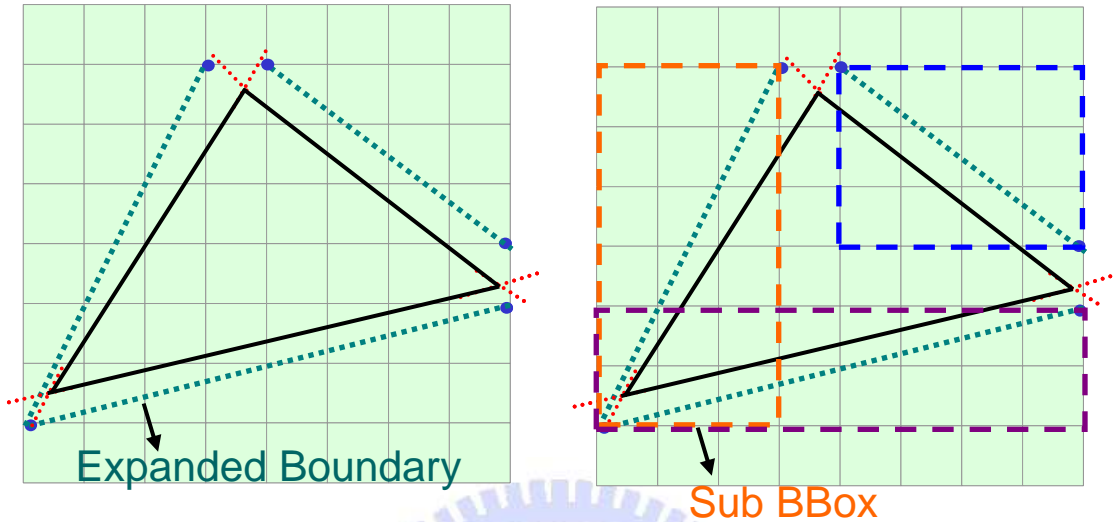


Figure 3-19 Expanded Boundary and Building Sub BBox

### 3.2.3 Elimination by Table Lookup

For the sub BBoxes, we can use the height and width of a sub BBox to look up a differential table through a converter shown in Fig. 3-20 to know the difference of the numbers of false-overlap tiles between two adjacent rows. An example is shown in Fig. 3-21.

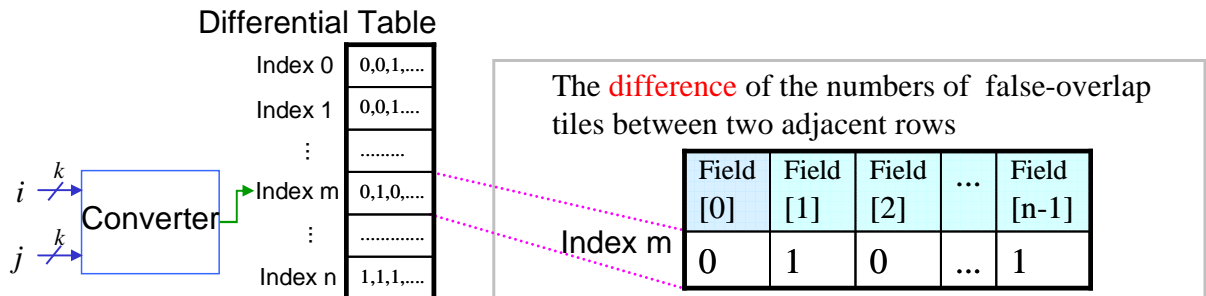


Figure 3-20 Using the width ( $i$ ) and height ( $j$ ) of a sub BBox to look up the differential table

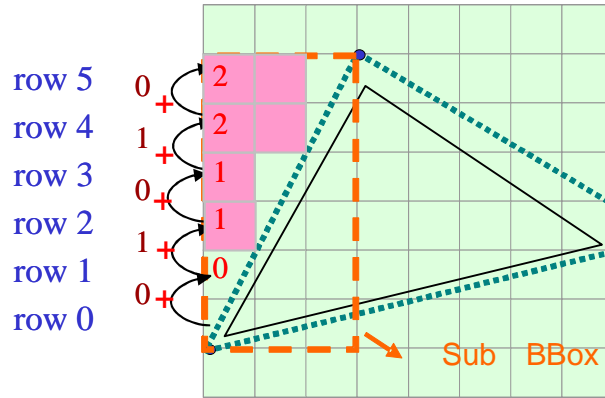


Figure 3-21 Example of eliminating false-overlap tiles with differential table

### Differential table

The differential table is pre-computed offline. Each field of an entry of the table records the difference of the numbers of false-overlap tiles between two adjacent rows. In the differential table, the first field of an entry means the difference between the row 0 and row 1 of a sub BBox, and so on. Here the row 0 is the first row in the sub BBox.

To reduce the number of entries of the differential table, we use three skills described as follows and apply  $i$  for antecedent and  $j$  for back tern to explain ratio :

(a) If  $i1:j1 = i2:j2$ , then these two ratios map to the same entry of the differential table.

Ex: 1:2 and 3:6 map to the same entry of table, as Fig.3-22(a) shown.

(b) If  $i1:j1 = j2:i2$ , these two ratios to the same entry of the differential table.

Ex: 2:1 and 1:2 map to the same entry of table, as Fig.3-22(b) shown.

(c) Adopt the approximation of the reciprocal of slope. Let the denominator  $d$  of the approximate ratio be a specific integer, such are 8, 16, 32, or 64, to approximate the original ratio. Then,  $\frac{0 \sim d-1}{d}$  ( $\frac{0 \sim 7}{8}, \frac{0 \sim 15}{16}, \frac{0 \sim 31}{32}$  or  $\frac{0 \sim 63}{64}$ ) may represent all

slopes, and thus the number of entries of the approximated table may be reduced to  $d$  (8, 16, 32, or 64). For example, 2:3, the ratio is  $\frac{2}{3}$ . We can use  $\frac{5}{8}$  to represent it when the denominator is 8. Since  $\left\lfloor \frac{2}{3} \times 8 \right\rfloor = 5$ , as shown in Fig. 3-19(c). And Table 3-1 shows the errors and and table sizes in different number of differential table entries.

Equation 3-1 is the mathematic equation of the inputs and outputs of the converter. Here  $i$  and  $j$  may excahnge to preserve that  $j$  is larger than  $i$ .

$$output_{index} = \left\lfloor \frac{input\_i}{input\_j} \times entry\_number \right\rfloor \quad (3-1)$$

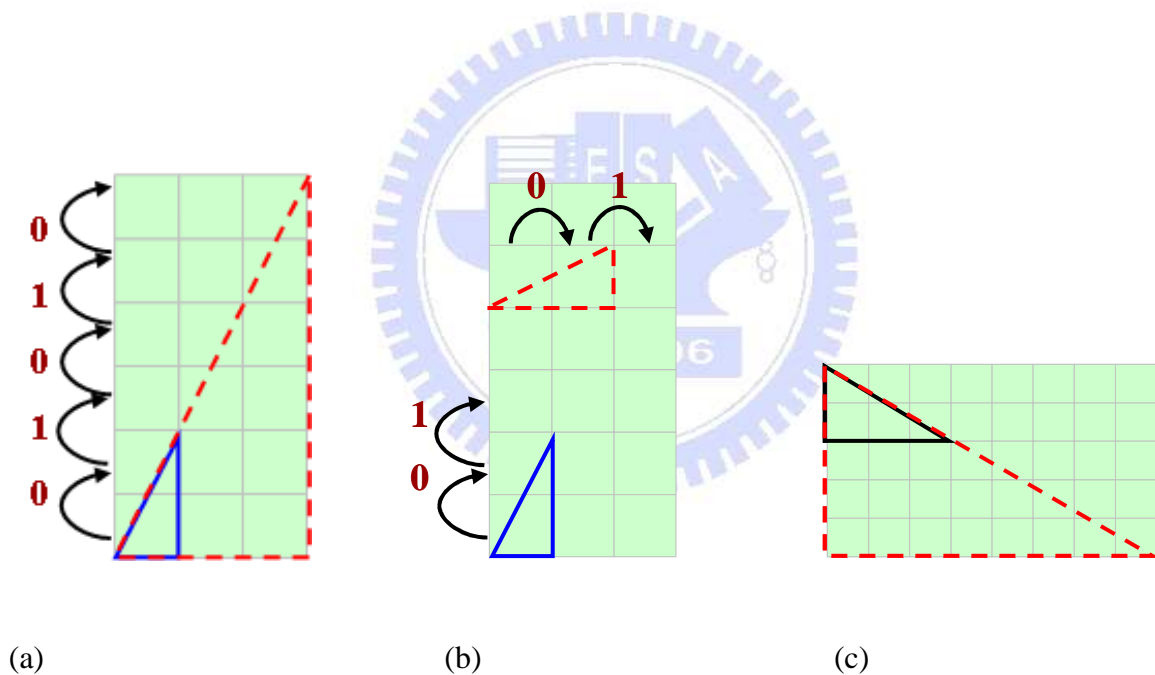


Figure 3-22 Example of differential table reduction. (a) 1:2 and 3:6 have the same differential of false-overlap tile pattern. (b) 2:1 and 1:2 have the same differential of false-overlap tile pattern. (c) 2:3, the ratio is  $\frac{2}{3}$ . We can use  $\frac{5}{8}$  to represent it when the denominator is 8.



Table 3-1 Error of different table entry. Screen size is 1200 x 1600, and tile size is 32 x 32

Number of Entry	Max error	Bits	Table size (bits)
8	0.12	8	56
16	0.06	15	240
32	0.03	31	992
64	0.01	63	4032

Because that there is no false-overlap tile in the first row of a sub box, the first bit ( Field[0] ) of the entry in a differential table is the number of false-overlap tiles in the second row of the sub BBox as Fig. 3-23 shown. We let the larger one of the height and width of the sub BBox be the specific entry numbe and adjust the samller one to be *index* which come from the  $output_{index}$  of equation 3-1. The Field[0] can be represent as equation 3-2. The second bit ( Field[1] ) of the entry is the difference of fasle-overlap tiles between row 1 and row 2. And the number of false-overlap tiles in row 2 is the floor of double of  $\frac{index}{entry\_number}$ . Thus, Field[2] can be represent as equation 3-3. The number of the false-overlap tiles in row *n* is the floor of n times of  $\frac{index}{entry\_number}$ . Therefore, Field[n-1] is as equation 3-4 shown.

$$\text{Field}[0] = \left\lfloor \frac{index}{entry\_number} \right\rfloor \quad 3-2$$

$$\text{Field}[1] = \left\lfloor \frac{index}{entry\_number} \times 2 \right\rfloor - \left\lfloor \frac{index}{entry\_number} \right\rfloor \quad 3-3$$

.

.

.

$$\text{Field}[n-1] = \left\lfloor \frac{index}{entry\_number} \times n \right\rfloor - \left\lfloor \frac{index}{entry\_number} \times (n-1) \right\rfloor \quad 3-4$$

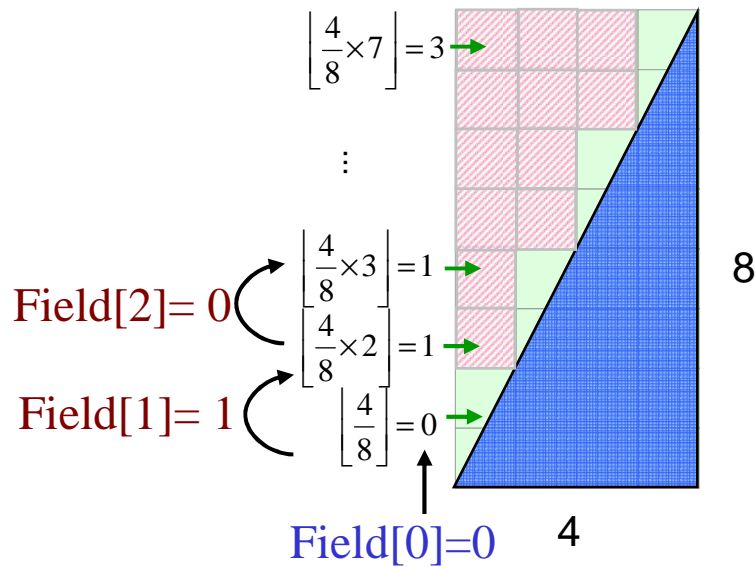
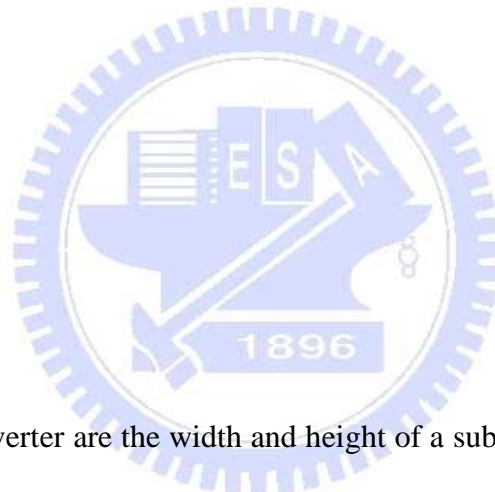


Figure 3-23 Difference of false-overlap tiles between adjacent rows



### Converter

The inputs of the converter are the width and height of a sub BBox, and the output is the index of the differential table. To reduce the complexity of circuit of the converter, we let  $i$  be the width of the sub BBox and  $j$  be the height, and exchange  $i$  and  $j$  to make  $j$  is larger than  $i$ . When  $i$  and  $j$  are equal, we know that the difference of nubmer of false-overlap tiles in adjacent rows is one without table lookup. The circuit is as Fig.3-24.

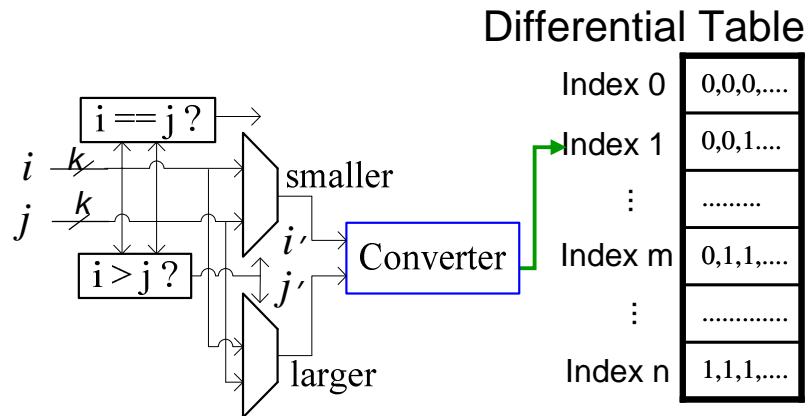


Figure 3-24 Circuit in front of Converter

Because the output of the converter is the index of the differential table, reducing the entries of the table also may simplify the complexity of the converter. The entry of the differential table records the difference of false-overlap tiles in adjacent rows for specific slope. And the differential table is pre-computed off-line. Table 3-2 is an example of the converter. The input  $i'$  and  $j'$  in Table 3-2 are the inputs of converter as shown in Fig.3-21. We can use approximate ratio to approach  $i'/j'$  and get the output  $t_{index}$  through the converter according to equation 3-1.

### False-overlap Tiles Elimination

In preserved region, let the vertex on the hypotenuse with the smaller edge of the width and height be point *Start*, the other one be point *End*. If the width ( $i$ ) of a sub BBox is smaller its than height ( $j$ ), we eliminate false-overlap tiles from point *Start* to point *End* row by row as Fig. 3-25(a) shown. Otherwise, we eliminate false-overlap tiles from point *Start* to point *End* column by column as Fig 3-25(b) shown. By the width and height of a sub BBox, we may look up the differential table and get the differeneec pattern of false-overlap tiles in rows.

Table 3-2 Inputs and output of converter

input $i'$	input $j'$	$i' / j'$	Approximate ratio	output <sub>index</sub>
1	2	0.50	4/8	4
1	3	0.33	2/8	2
⋮	⋮	⋮	⋮	⋮
1	10	0.10	0	0
2	3	0.67	5/8	5
2	4	0.5	4/8	4
⋮	⋮	⋮	⋮	⋮
2	10	0.20	1/8	1
⋮	⋮	⋮	⋮	⋮
8	9	0.89	7/8	7
8	10	0.80	5/8	5

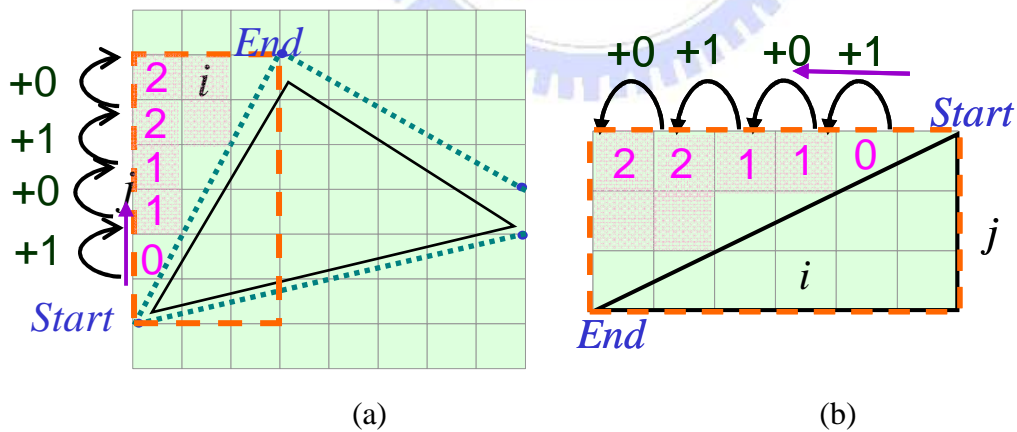


Figure 3-25 Direction for eliminating false-overlap tiles

With the direction for eliminating false-overlap tiles and the difference pattern of false-overlap tiles in rows (columns), we can accumulate the bits of difference pattern to get the number of false-overlap tiles in each row (column) from the second row (column). For

example, there is a primitive (with black line) in Fig. 3-26, and its width is 8 and height is 10. Let the table size be 8. Through our converter, we can use the entry which record the false-overlap tile with approximate difference of  $6/8$  to eliminate the false-overlap tiles. We make the first row in the primitive be row 0, and so on. The initial value of false-overlap\_tile is zero for row 0 and accumulate the value of field in the entry to false-overlap\_tile. The value of accumulated false-overlap\_tile is the number of false-overlap tiles in each row.

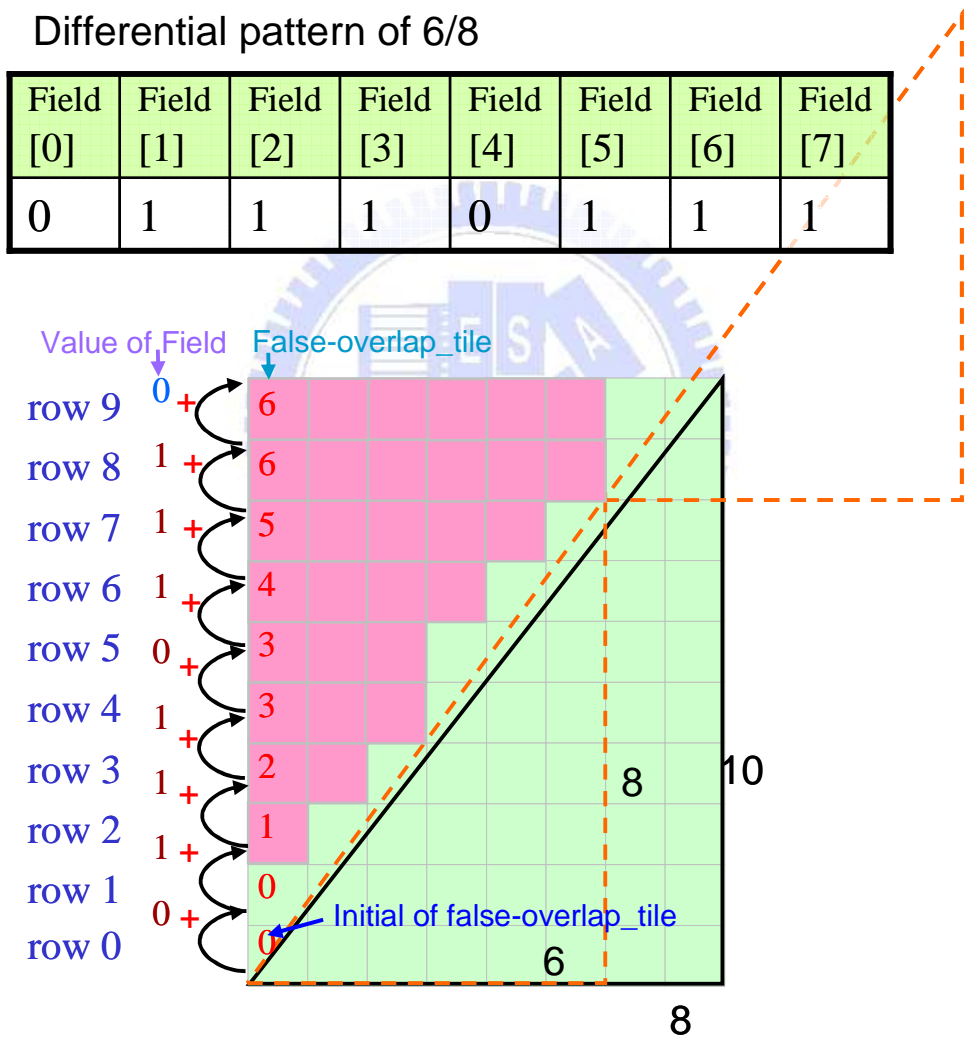
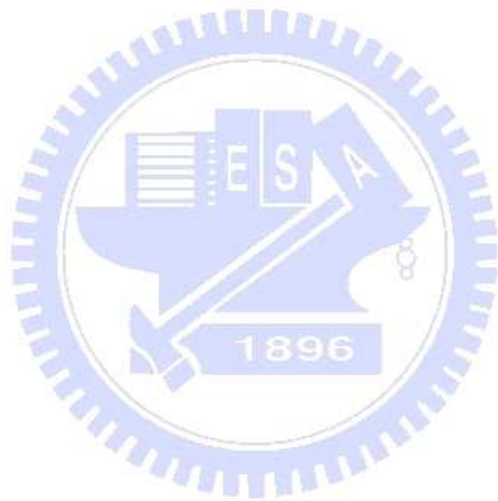


Figure 3-26 Example of eliminating rows larger the differential pattern



## Chapter 4 Evaluation Results and Discussion

In this chapter, we first describe our evaluation environment and the characteristics of the input frame data (in section 4.1 and 4.2). Then, we show and analyze the simulation results of memory requirement and execution time during rendering of each method: Bounding Box test, LET, iterative division test of Intel, and our exact and approximate methods in section 4.3. In the last section, we briefly discuss and summarize our conclusion from the results.

### 4.1 Evaluation environment

Figure 4-1 shows the architecture of ATTILA simulator and in which stage we dump the coordinates of transformed primitives from triangle setup of the ATTILA GPU simulator for the input of our simulation. We implemented a behavioral simulator of our architecture in C++, and modified ATTILA simulator [13] to output coordinates information to a tracefile. The benchmarks chosen are DOOM3 and QUAKE4 [14], modern graphics applications, and resolutions are 320x240, 640x480, 1280x1024 and 1600x1200 in frame 30, 60, 90, 120, 150, 180, 210, 240, 270, and 300. Fig. 4-2 and Fig.4-3 show one frame in DOOM3 and QUAKE4, respectively. The trace file outputted from ATTILA simulator contains the coordinates in frames. Our simulator reads the tracefile and evaluates storage size and correct rate of primitive lists.

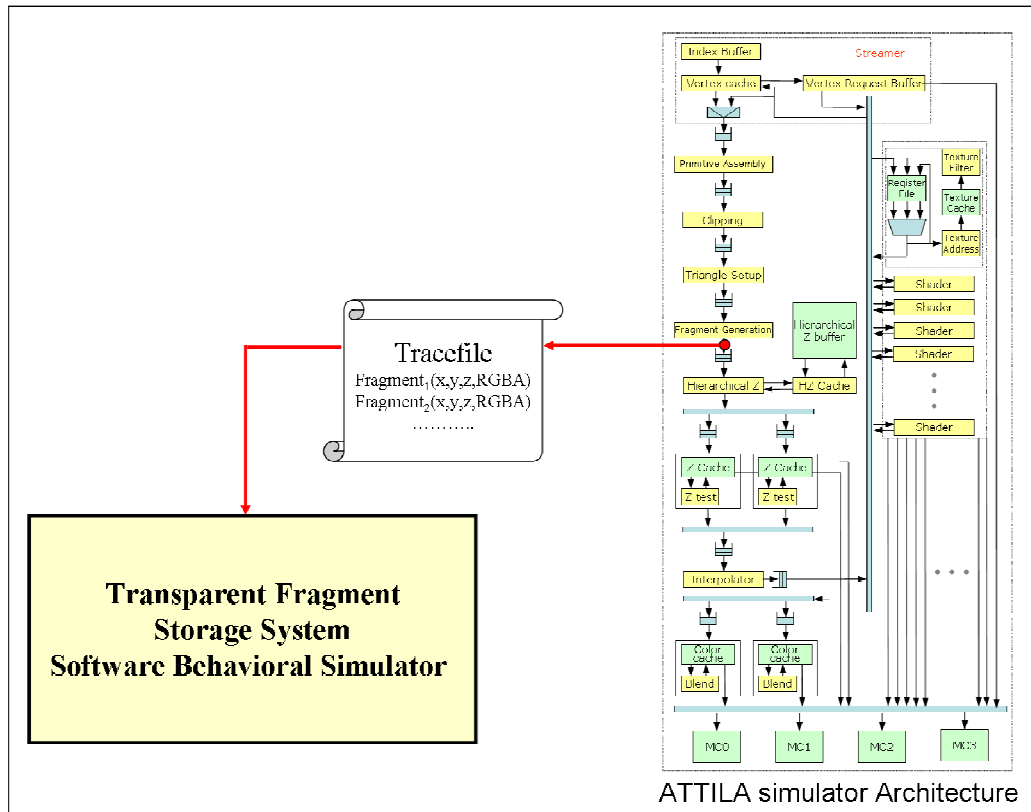


Figure 4-1 simulation flow and ATTLA architecture [13]

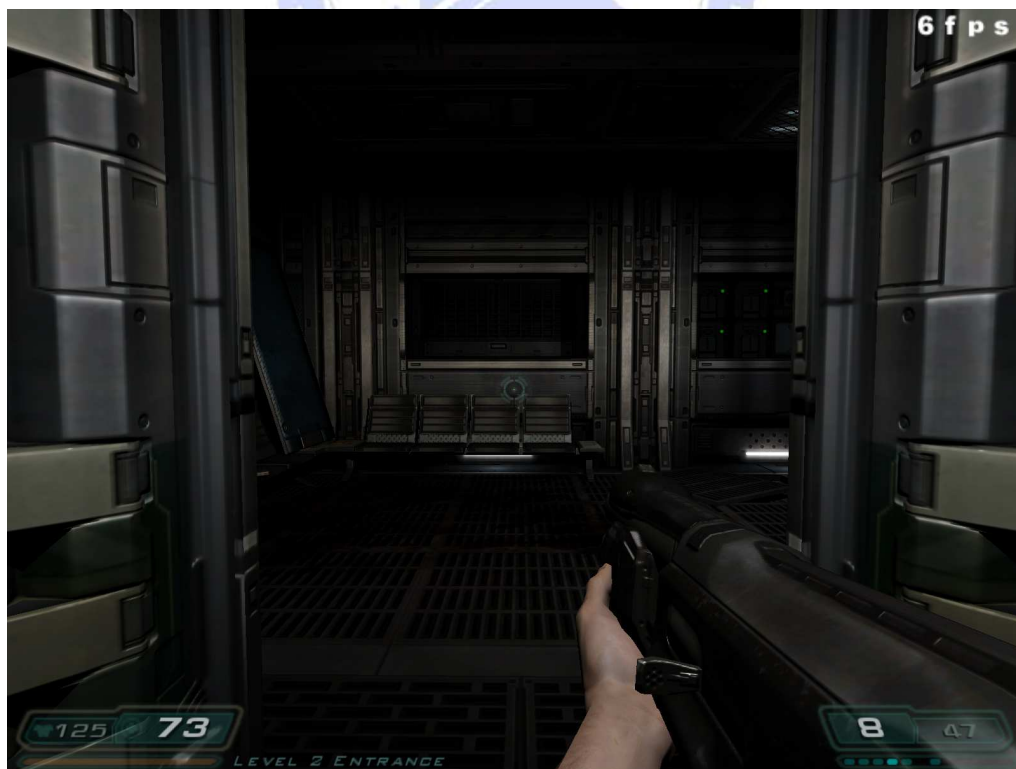


Figure 4-2 Frame 30 in DOOM3





Figure 4-3 Frame 30 in QUAKE4

## 4.2 Test frame data

In this section, we provide statistics of Doom3 and Quake4 in different screen sizes of frames, for frame30, 60, 90, 120, and 150, as shown in Table 4-1 and Table 4-2. In Table 4-1 and Table 4-2, the second row indicates the average number of tiles covering in a primitive; the third row shows the maximum number of tiles covering in a primitive; the fourth row shows the total number of tiles in all BBoxes; the fifth row shows the number of actually overlapped tiles; the sixth row brings the percentage of tiles really be rendered; the seventh row shows the average height of primitives in number of tiles. The last row shows the average width of primitives in number of tiles. And Table 4-3 shows Average number of different operations per right triangle of each algorithm in various tests.

Table 4-1 Statistics of test frames on Doon3

Screen size	320x640	640x480	1280 x 1024	1200x1600
Actual Tile Coverage of Primitives (Avg)	2.87	6.34	19.00	25.39
Maximum Tile Coverage of Primitives	70	300	1280	1850
Total Number of Tiles in all BBoxes	24428	54047	161864	216362
Number of Actually Overlapped Tiles	16265	30076	71860	91792
Tile correct ratio	66.58%	55.65%	44.40%	42.43%
Average Height of Primitives in number of tiles	1.8	2.7	4.8	5.4
Average Width of Primitives in number of tiles	1.5	2.0	3.1	3.6

Table 4-2 Statistics of test frames on Quake 4

Screen size	320x640	640x480	1280 x 1024	1200x1600
Actual Tile Coverage of Primitives (Avg)	7.45	24.91	94.81	135.64
Maximum Tile Coverage of Primitives	80	300	1280	1850
Total Number of Tiles in all BBoxes	381553	1276076	4857531	6949631
Number of Actually Overlapped Tiles	125220	262100	659722	862638
Tile correct ratio	32.82%	20.54%	13.58%	12.41%
Average Height of Primitives in number of tiles	2.4	3.8	7.2	8.3
Average Width of Primitives in number of tiles	2.4	4.1	7.3	8.9

Table 4-3. Average number of different operations per right triangle of each algorithm in various tests.

Benchmark	Cross Product Test (CPT)				Edge Walk Test(EWT)				Count X Ratio (CXR)						Divide-and-conquer and table lookup	
	Doom 3		Quake 4		Doom 3		Quake 4		Doom 3			Quake 4			Doom 3	Quake 4
Operation	MUL	SUB	MUL	SUB	ADD	SUB	ADD	SUB	Add	SUB	DIVI	Add	SUB	DIVI	ADD	ADD
320x240	6	3	6	3	4	6	4	6	4	3	1	5	3	1	1	1
640x480	8	4	10	5	6	9	8	12	5	3	1	6	3	1	1	1
1280x1024	12	6	16	8	10	15	14	21	7	3	1	10	3	1	1	1
1600x1200	12	6	18	9	10	15	16	24	8	3	1	11	3	1	1	1

## 4.3 Simulation results

In this section, we show the simulation results of time complexity, primitive list, and primitive list correct ratio. We compare the results of our design with the method of BBox test [4, 6-8], Linear Edge Function Test (LET) [11] and Iterative Division Test (IDT) [13].

### 4.3.1 Correct Rates of Different Entry Size of Differential Table for our Approximation Method

Fig. 4-4 and Fig.4-5 show the relationship of the entry sizes of differential table with correct rate for DOOM3 and QUAKE4, respectively. More differential table entries bring more false-overlap tile detection. As Fig. 4-4 shown, in DOOM3, screen size of 320x240 needs 4 entries for differential table, 640x480 needs 8 entries, and there are 16 entries enough are for 1280x1024 and 1600x1200. However, in QUAKE4 as Fig. 4-5 shown, 8 entries are enough for 320x240, 16 for 640x480, and 640x480 and 1200x1600 needs 32 entries for differential table. As the screen size becomes larger, the primitives become larger and need more table entry to remove false-overlap tiles. Since more table entry may remove false-overlap tile more precisely. Furthermore, the primitives in QUAKE4 are larger than in DOOM3, and thus the approximate method needs more table entry to get higher correct rate at the same screen size.

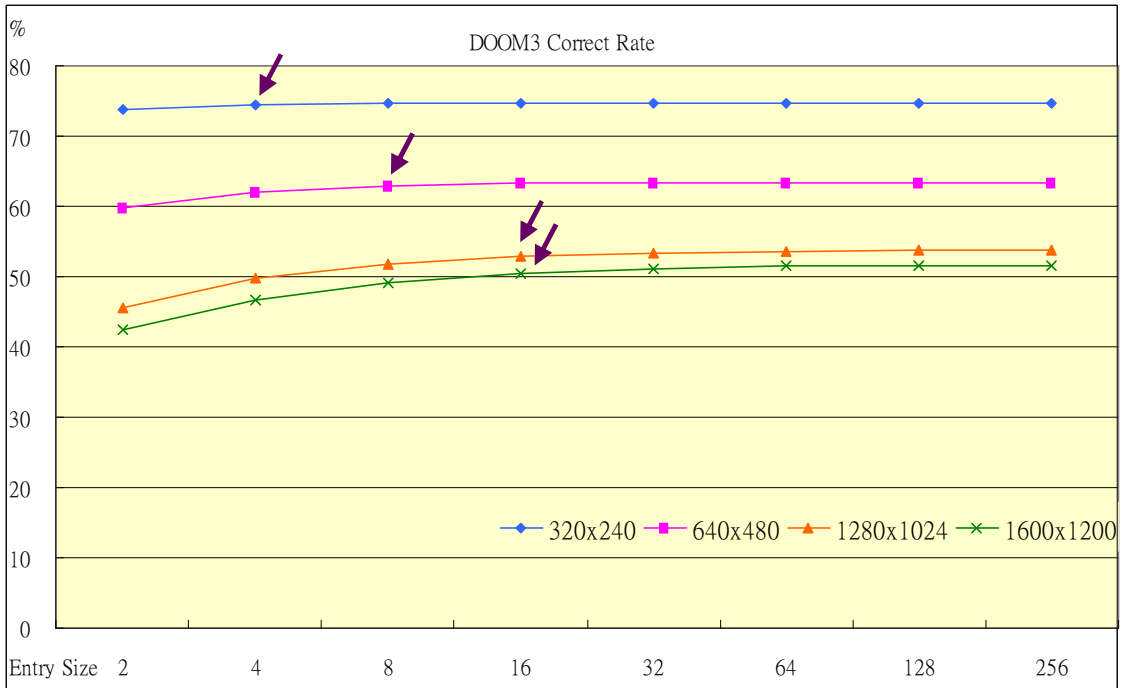


Figure 4-4 Correct rate with entry size for DOOM3

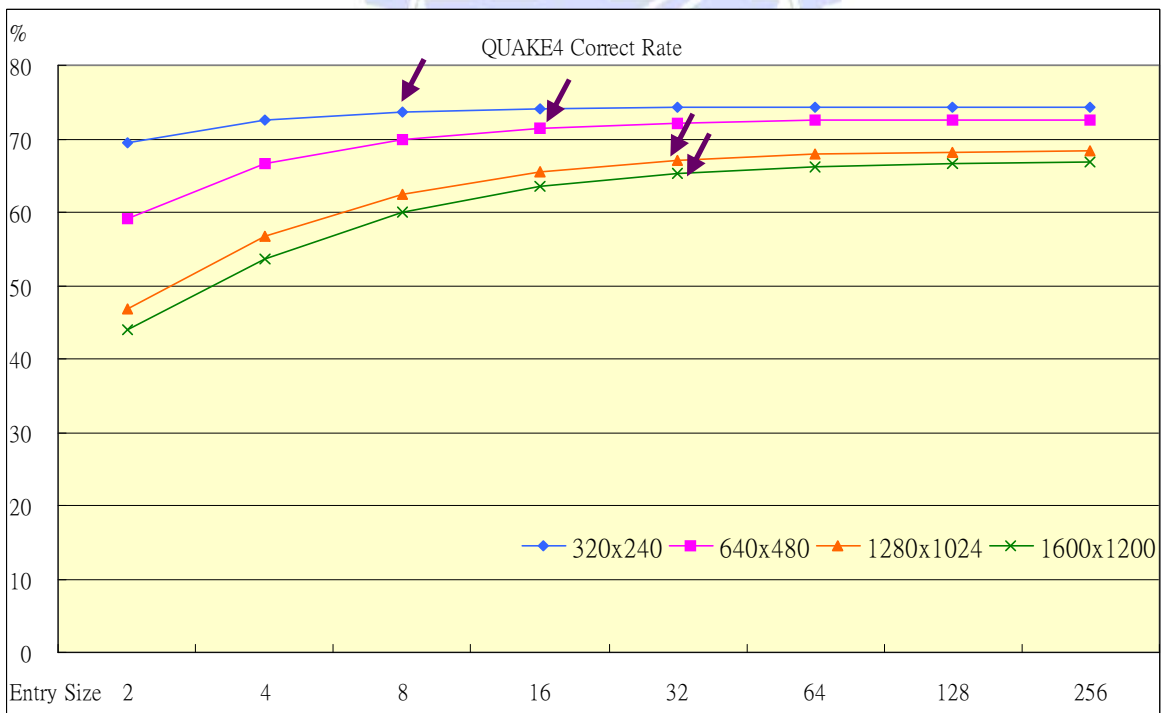


Figure 4-5 Correct rate with entry size for QUAKE4

### 4.3.2 Storage Size of Primitive Lists

In order to evaluate our methods, we compare our storage size with the method of BBox test [4, 6-8], Linear Edge Function Test (LET) [11] and Iterative Division Test (IDT) [13]. We list the amount of primitive lists of each method both in Figure 4-6 and Figure 4-7 for DOOM3 and QUAKE4. The first and orange bar is the primitive lists really rendered, and it also the results of our exact methods CPT, EW, and CXR. The second bar is the amount of primitive list with BBox test. The third bar is the primitive list with LET, and the fourth to sixth bars are IDT with one to three times of iterative. Since the results of them are close to our approximate method, we list the results of IDT with one to three times of iterative to compare and discuss. Furthermore, the seventh to fourteen bars are the primitive lists in for own approximate method with 2, 4, 8, 16, 32, 64, 128 and 256 entries of differential table.

Although the primitive lists in IDT are the fewest among BBox, LET, IDT and approximate method with different entries in Fig. 4-6 with one iterative, they needs more iterative when the primitives are larger as Fig. 4-7 shown. In Fig. 4-7, IDT needs three iterative to get the results which are close to approximate method. In the other words, a primitive can be divided up to ten iterative and get  $3^3$  smaller primitives to process.

Fig. 4-8 shows different iterative and storage size of the primitive list with IDT, and Fig.4-9 shows different iterative and correct rate of primitive with IDT. Here, we use correct rate to represent the percentage of the primitive lists which are really rendered. From Fig.4-8 and Fig.4-9, we can see that the IDT can get less storage size of primitive lists and higher correct rate with more iterative, but it also costs more time with power of three. Although the LET produces the same storage size of the primitive lists comparing to our approximate method in Fig. 4-6 and Fig. 4-7, it needs floating subtraction, multiplication and division.

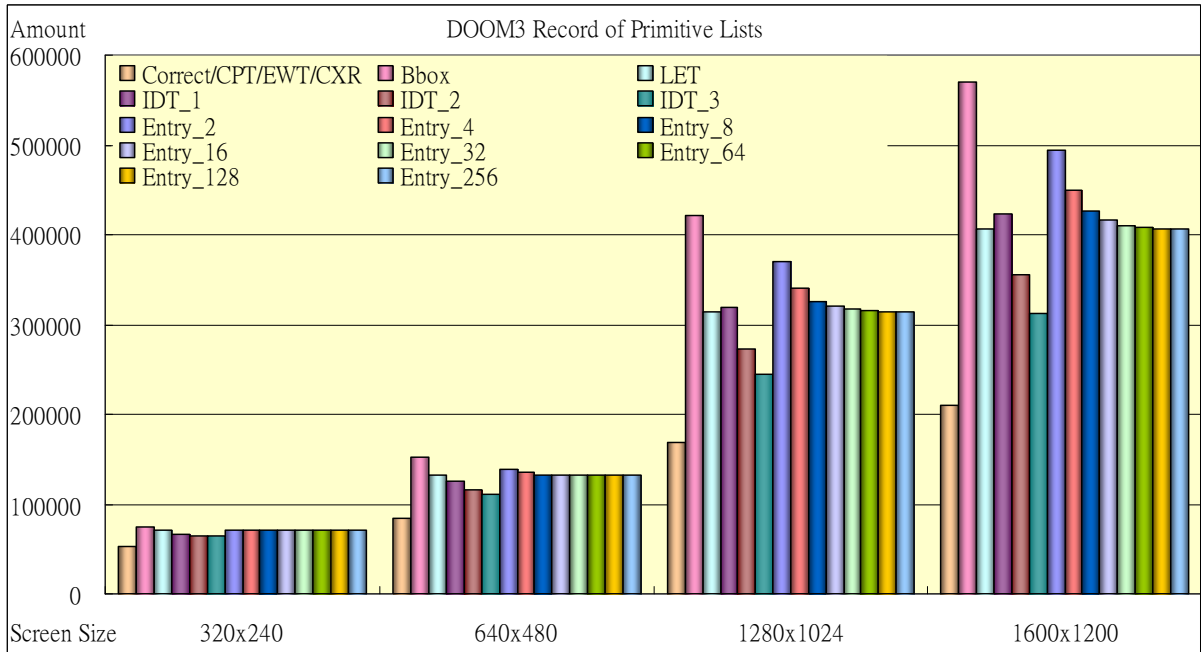


Figure 4-6 Amount of primitive lists for Doom3

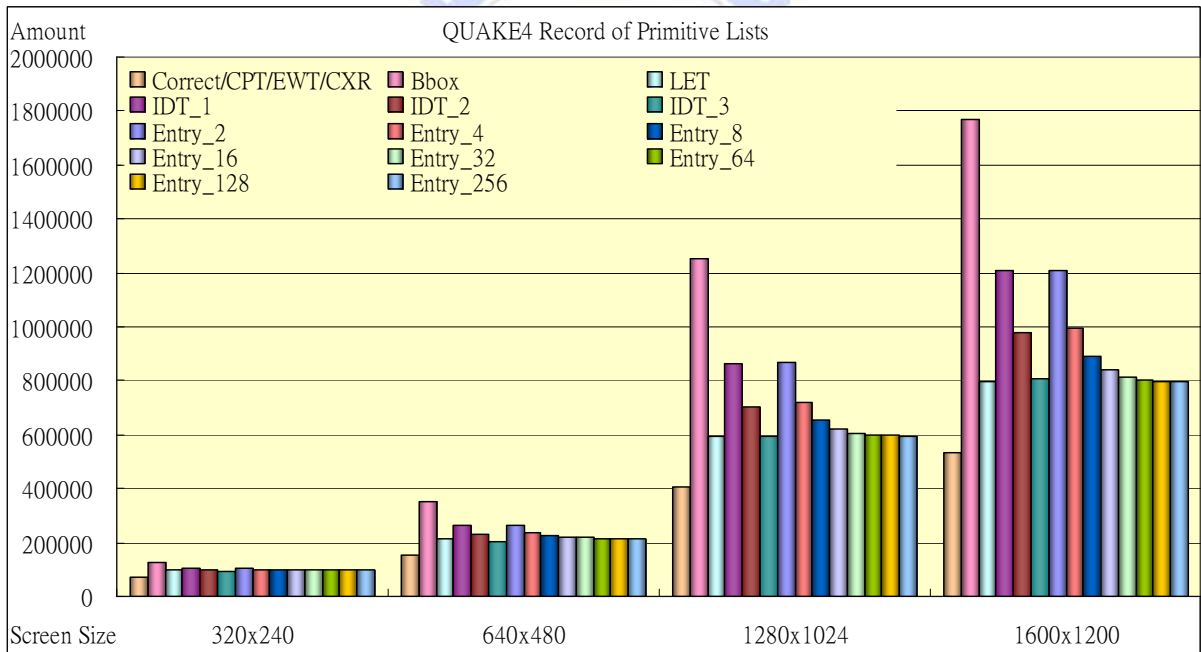


Figure 4-7 Amount of primitive lists for Quake4

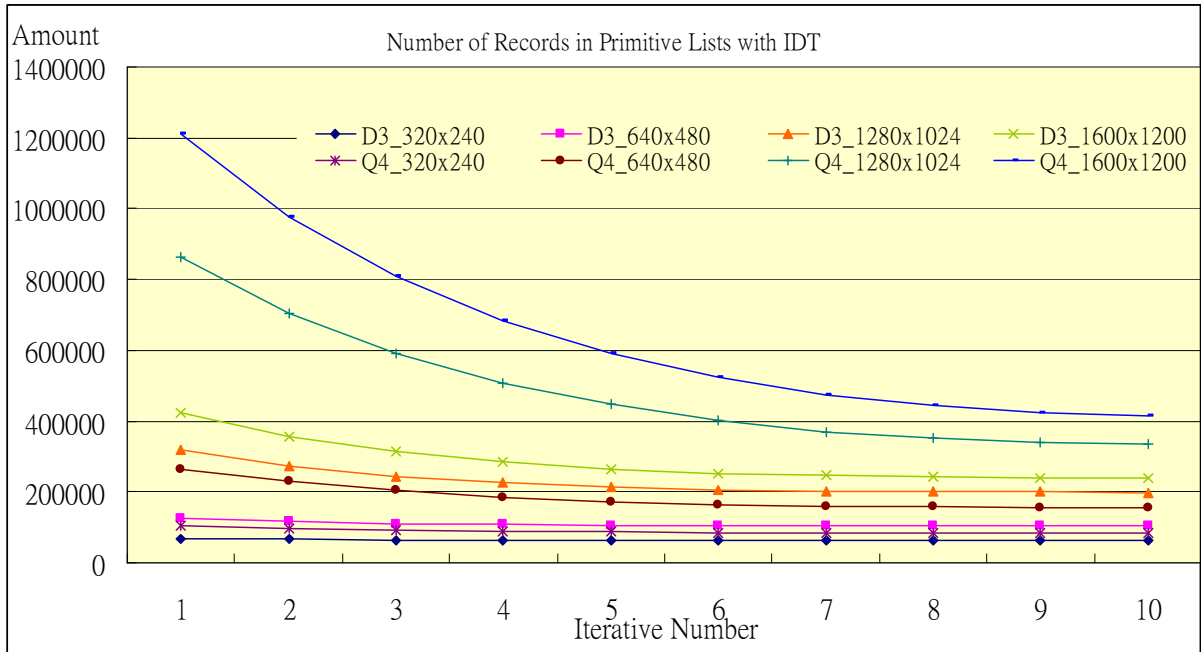


Figure 4-8 Number of Records in Primitive Lists with Iterative Division Test

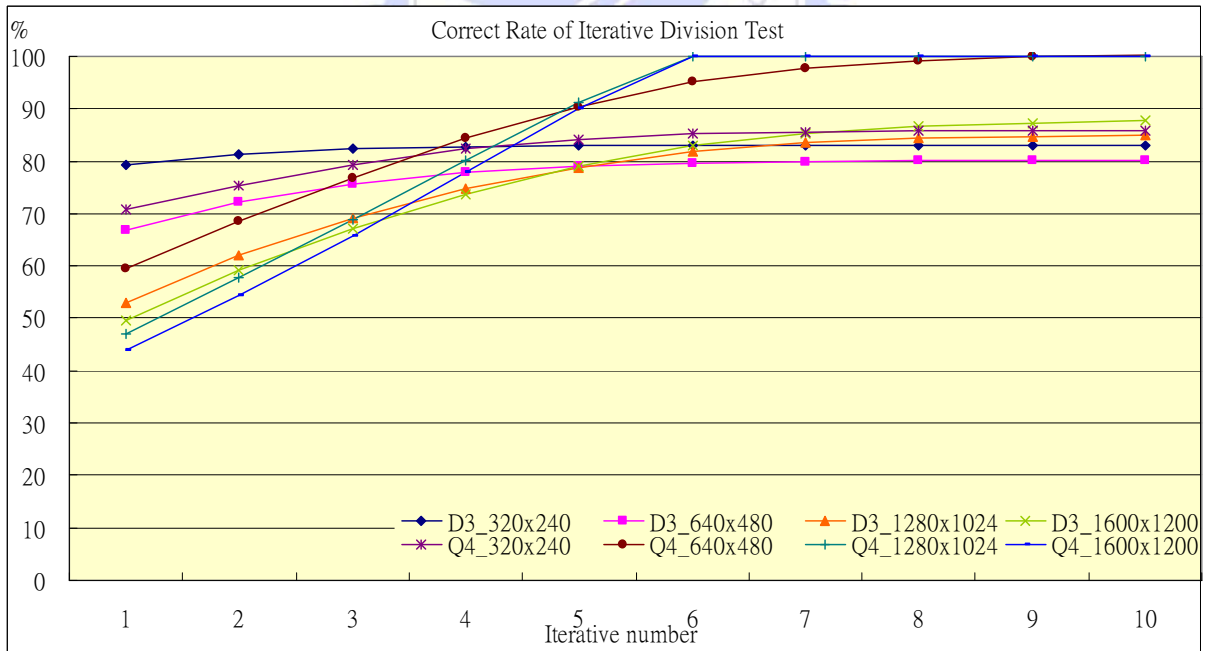


Figure 4-9 Correct Rate of Iterative Division Test

Differential table with more entries can get fewer primitive lists and higher correct rate.

The difference of correct rate between 8 and 64 entries of differential table is up to 5% as Fig.4-10 and Fig. 4-11 shown.

$$\text{Correct Rate} = \frac{\#records\_TestMethod}{\#records\_RealRender} \quad (4-1)$$

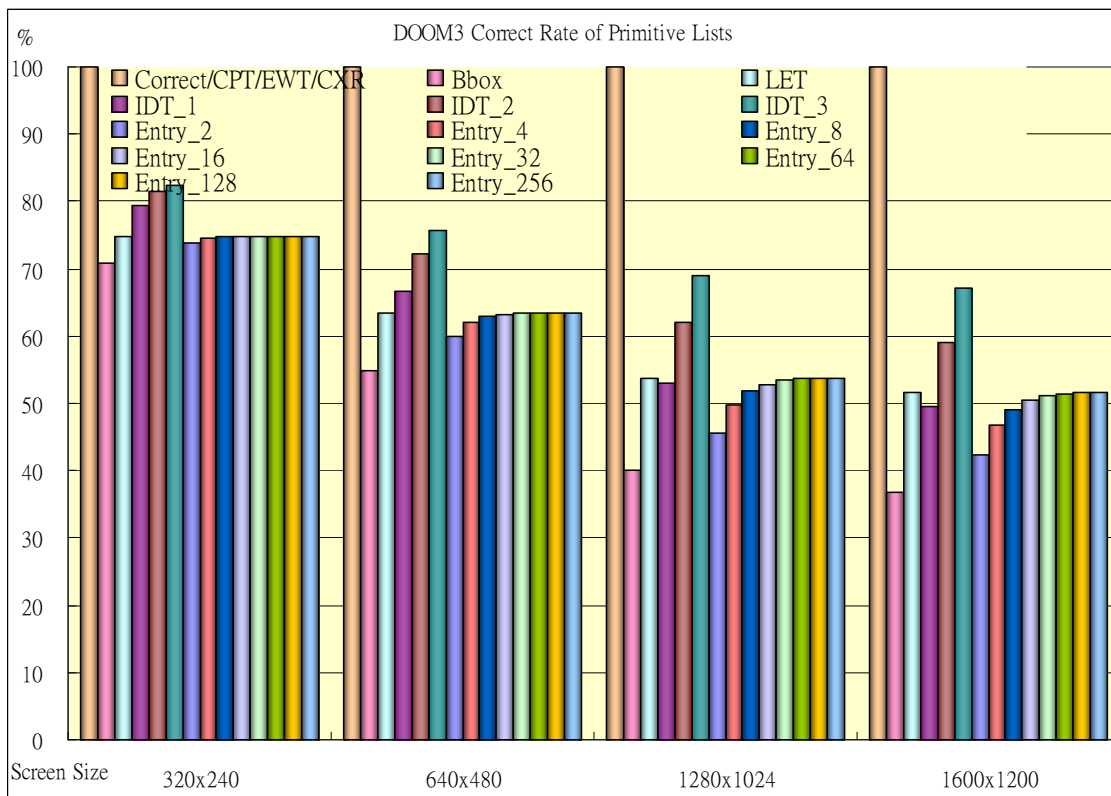


Figure 4-10 Correct rate of primitive lists for DOOM3



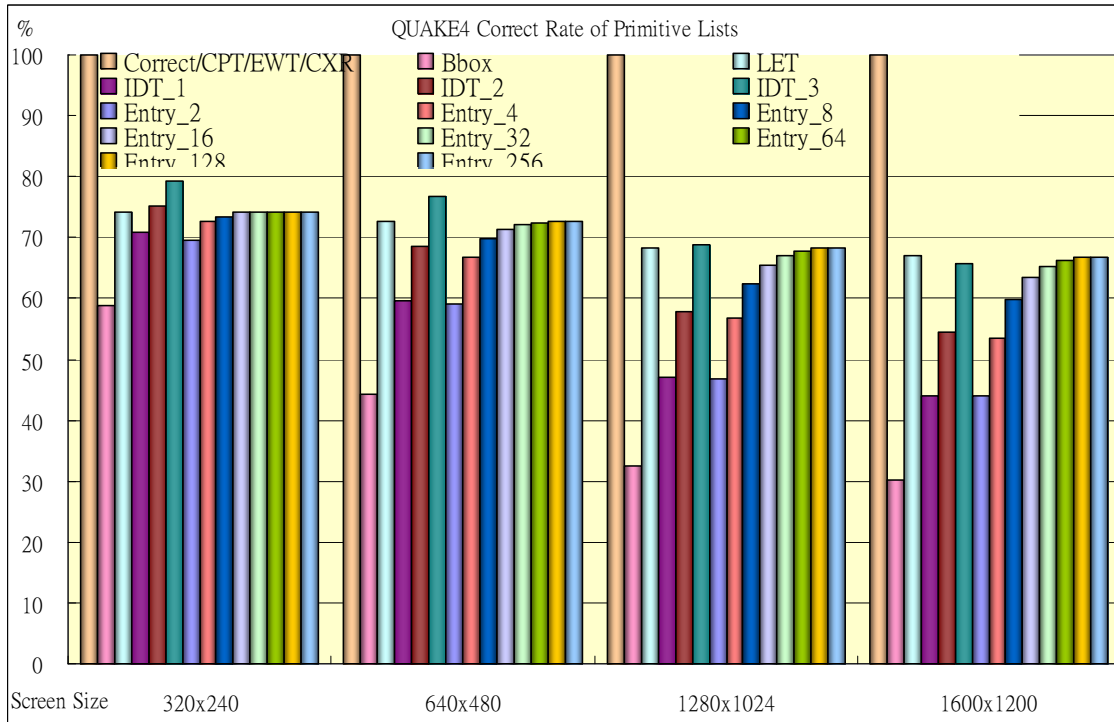


Figure 4-11 Correct rate of primitive lists for QUAKE4

For larger size and primitive, our approximation method may eliminate more false-overlap tiles as Fig. 4-12 and Fig.4-13 shown.

$$\text{Reduction Rate} = \frac{\#records\_BBox - \#records\_TestMethod}{\#records\_BBox} \quad (4-2)$$

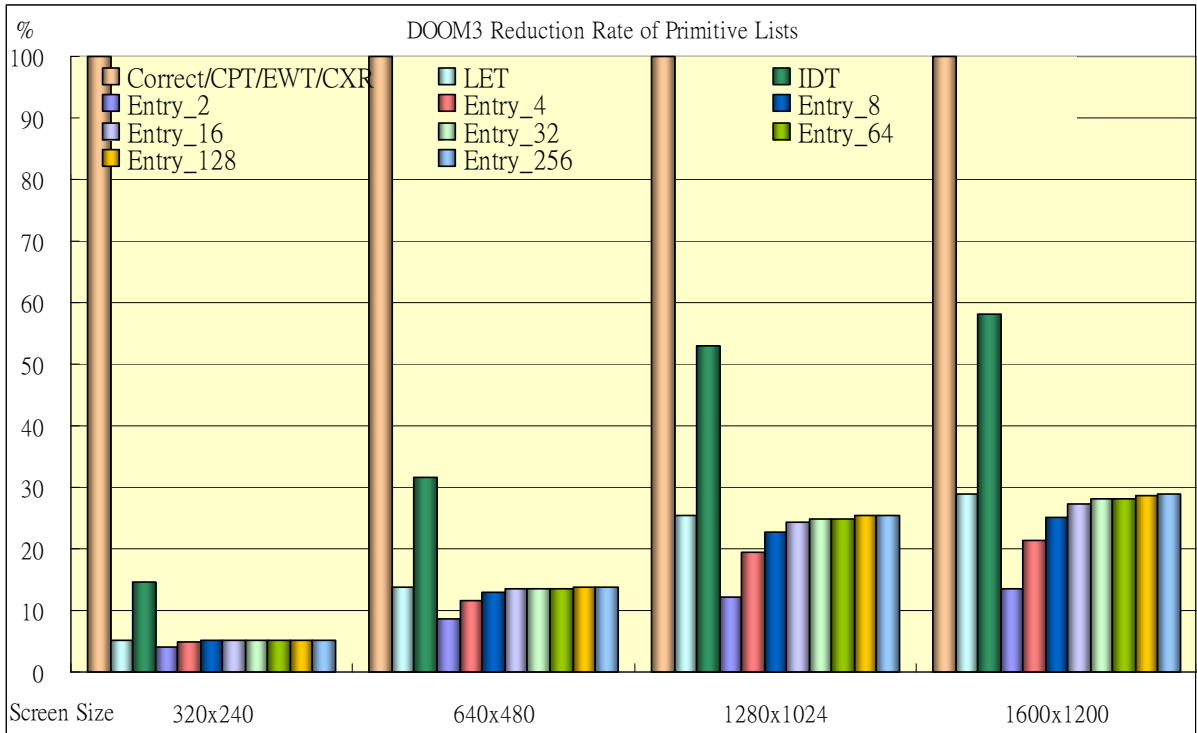


Figure 4-12 Reduction Rate of Primitive Lists for DOOM3

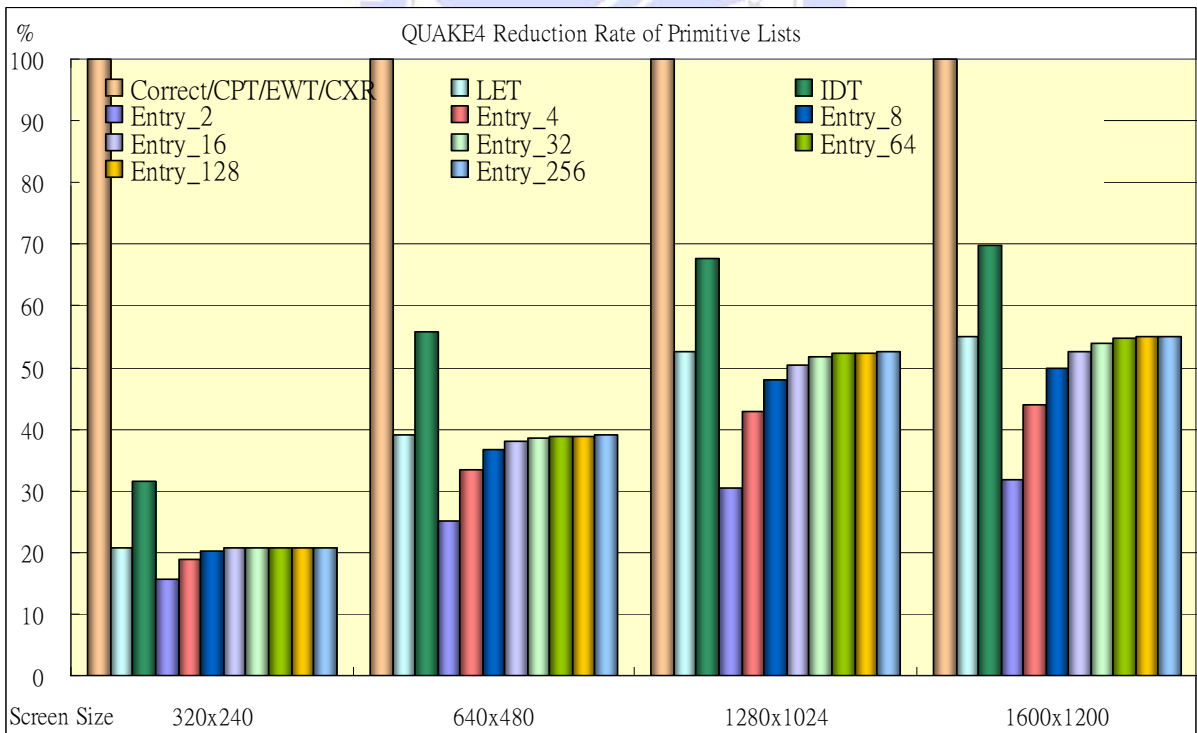


Figure 4-13 Reduction Rate of Primitive Lists for QUAKE4

### 4.3.3 Time complexity

The time complexity of our methods and LET are shown in Fig. 4-14. IDT divides the BBox by adding middle points on each edge of primitive, so it needs three additions and shifts before next iterative.

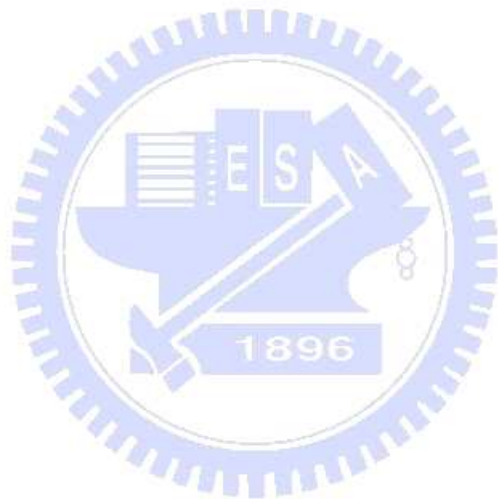
Although the approximation method has additional converter and differential table, it has fewest computations in false-overlap tiles and the correct rate is not worse than current design. In the exact methods, they all have the same correct rate, 100%. However, the CPT has the highest computing time.

If we just require the correct without considering hardware cost, CPT and EWT are good choices for the benchmark with small primitive. And CXR is good for larger primitive. Since the operation for each row tile in CXR only needs an addition. Comparing to exact methods, our approximate can reduce primitive lists without adding too much hardware cost.

Table 4-4 Time complexity of our methods and relate work

	Linear Edge Function Test (LET)	Cross Product Test (CPT)	Edge Walk Test (EWT)	Count X Ration (CXR)	Expand Boundary and Look up table
Pre-processing	$\frac{l}{2} \cdot ( x_B - x_A  +  y_B - y_A )$ $\frac{l}{2} \cdot ( x_C - x_B  +  y_C - y_B )$ $\frac{l}{2} \cdot ( x_A - x_C  +  y_A - y_C )$			1. Compute the initial of x_ration 2. For reciprocal of slope	1. Expand Boundary 2. Look up converter table 3. Look up different table
Operation in each row tile	For tile: $E_{Lab}(x_{CS}, y_{CS})$ $E_{Lbc}(x_{CS}, y_{CS})$ $E_{Lca}(x_{CS}, y_{CS})$	Cross product- $\overline{SE} \times \overline{ST}$	Compute which tile is on the edge of primitive	x_ration = x_ration + 1/slope*(row_index-1)	Accumulate
Operation for each sub right triangle	Sub:6 Mul:3 Add:3 Divi:3			Sub :3 Add :2 Divi :2	Look up table
Operation for each row tile	Sub:3 Mul:2	Mul :2 Sub :1	Sub :3 Add :2	Add : 1	Add:1
Operator	most	medium	medium	medium	least
Close rate	50~70%	100%	100%	100%	50~75%

P.S.  $E_{Lab}(x, y) = (x - X) \cdot dY - (y - Y) \cdot dX$ .



# Chapter 5 Conclusion and Future Work

## 5.1 Conclusion

In this thesis, we propose exact and approximation methods to eliminate false-overlap tiles. In the exact methods, CPT, EWT, and CXR, all of them may eliminate all false-overlap tiles in a BBox. However, they need a lot of computations and hardware cost than our approximate method. With the pre-computed differential table, the approximate method can eliminate most of the false-overlap tiles in a BBox without complex computation. And our approximate method can eliminate 70% false-overlap tiles without complex computing. In DOOM3 and QUAKE4, the correct rate is up to 75% comparing to BBOX, LET and IDT.

Our exact methods, CTP, EWT and CXR, can product really work primitive lists which are really rendering in rasterizer. And CTP and EWT are suitable for benchmark with small primitives, like DOOM3. CXR is good for benchmark with large primitives, like QUAKE4. However, they all needs more hardware cost than our approximation method. Although, approximation method cannot provide 100% correct rate of primitive lists, it can look up differential table to get false-overlap tiles without complex computations.

## 5.2 Future work

In our observation, there are patterns of false-overlap tiles in rows. The patterns also can be employed to pixels in tile with a repaired table to assist the differential table. Then we can eliminate false-overlap tiles without aligning the primitive vertex to tile vertex, and can eliminate all false-overlap tiles in the BBox.

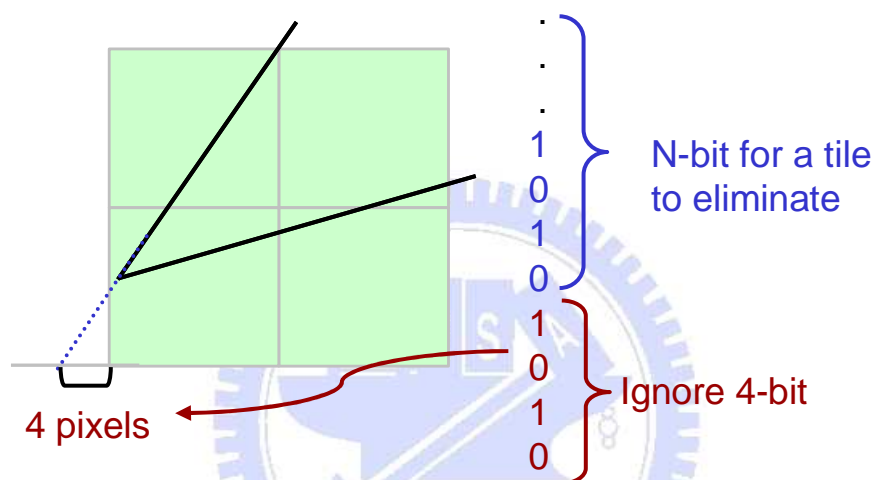


Figure 5-1 Use pixel pattern to eliminate false-overlap tiles

## References

- [1] “PowerVR. 3D Graphical Processing (Tile Based Rendering - The Future of 3D),” white paper, Imagination Tech. Corp., 2000.
- [2] “ARM Mali 3D Graphics System Solutions,” white paper, ARM Corp., Dec. 2006.
- [3] (2009) PowerVR SGX Series5XT Graphics IP Core Family, [Online]. Available: [http://www.imgtec.com/powervr/sgx\\_series5XT.asp](http://www.imgtec.com/powervr/sgx_series5XT.asp)
- [4] “Imageon 3D 238x White Paper,” white paper, ATi Corp., 2005.
- [5] E. Sorgard, B. Ljosland, J. Nystad, M. Blazevic, F. Langtind, “Method of and apparatus for processing graphics,” U.S. Patent 2007/0146378 A1, Jun. 28, 2007.
- [6] E. Hsieh, V. Pentkovski, and T. Piazza, “ZR: A 3D API Transparent Technology for Chunk Rendering,” In *Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture MICRO-34*, 2001.
- [7] M. Chen, G. Stoll, H. Igehy, K. Proudfoot, and P. Hanrahan, “Simple Models of the Impact of Overlap in Bucket Rendering,” In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 105–112, Lisbon, Portugal, 1998, ACM Press.
- [8] M. Cox and N. Bhandari, “Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC,” In *Proc. 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34, ACM Press, 1997.
- [9] I. Antochi, B.H.H. Juurlink, S. Vassiliadis, and P. Liuha, “Efficient Tile-Aware Bounding-Box Overlap Test for Tile-Based Rendering” , *Proceedings of the 2004 International Symposium on System-on-Chip 2004, Tampere, Finland*, November 2004, pp. 165-168.

[10] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proc. 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 17–20. ACM Press, 1988.

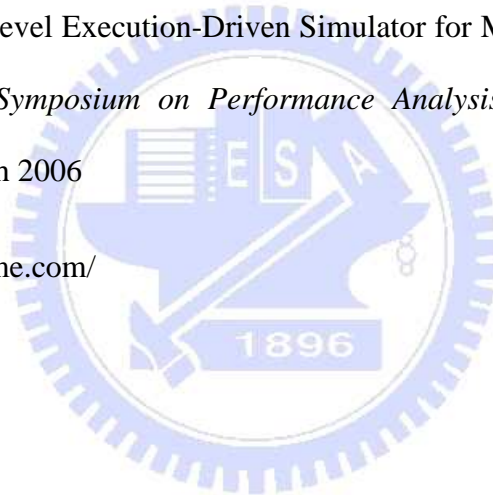
[11] Stephen Junkins, Oliver A. Heim, Lance R. Alba, “Methods and apparatuses for a polygon binning process for rendering”, U.S. Patent US 6,975,318 B2/US 7,167,171 B2

[12] Bresenham's line algorithm on Wiki:

[http://en.wikipedia.org/wiki/Bresenham%27s\\_line\\_algorithm](http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)

[13] Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa, “ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures”, *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006)*, March 2006

[14] <http://www.quake4game.com/>





## Appendix A. Simulation Test Frame Images



Figure A-1 frame30



Figure A-2 frame60



Figure A-3 frame90



Figure A-4 frame120



Figure A-5 frame150

